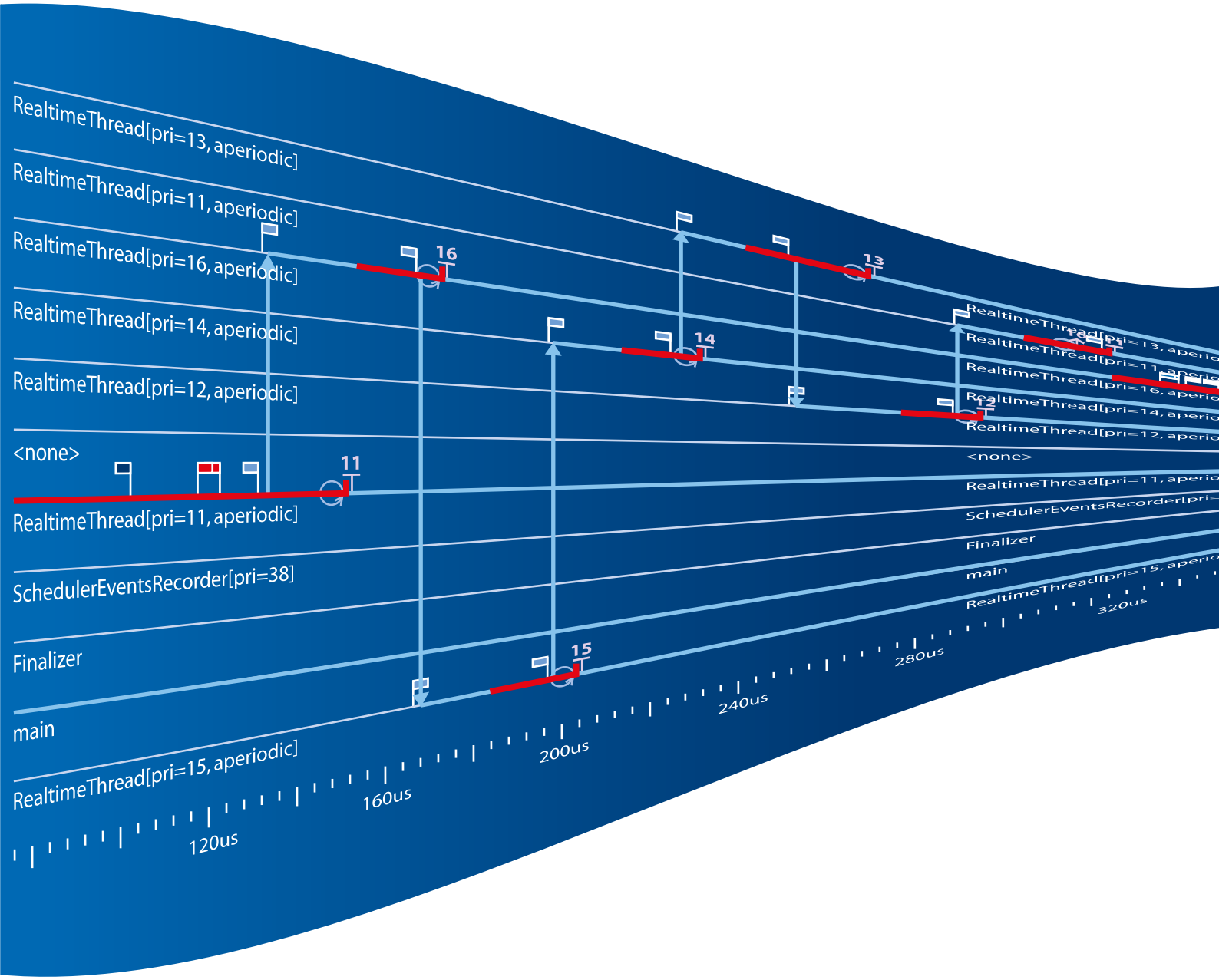
aicas realtime

JamaicaVM 6.2 User Manual

Java Technology for Critical Embedded Systems



JamaicaVM 6.2 — User Manual

Java Technology for Critical Embedded Systems

aicas GmbH

JamaicaVM 6.2 — User Manual: Java Technology for Critical Embedded Systems

JamaicaVM 6.2, Release 5. Published November 22, 2013.
©2001–2013 aicas GmbH, Karlsruhe. All rights reserved.

No licenses, expressed or implied, are granted with respect to any of the technology described in this publication. aicas GmbH retains all intellectual property rights associated with the technology described in this publication. This publication is intended to assist application developers to develop applications only for the Jamaica Virtual Machine.
Every effort has been made to ensure that the information in this publication is accurate. aicas GmbH is not responsible for printing or clerical errors. Although the information herein is provided with good faith, the supplier gives neither warranty nor guarantee that the information is correct or that the results described are obtainable under end-user conditions.

aicas GmbH
Haid-und-Neu-Straße 18
76131 Karlsruhe
Germany
phone +49 721 663 968-0
fax +49 721 663 968-99
email info@aicas.com
web http://www.aicas.com

aicas incorporated
6 Landmark Square, Suite 400
Stamford CT 06901
USA
phone +1 203 359 5705
email info@aicas.com
web http://www.aicas.com

aicas GmbH
9 Allee de l'Arche
92671 Paris La Defense
France
phone +33 1 4997 1762
fax +33 1 4997 1700
email info@aicas.com
web http://www.aicas.com

Java and all Java-based trademarks are registered trademarks of Oracle America, Inc. All other brands or product names are trademarks or registered trademarks of their respective holders.
ALL IMPLIED WARRANTIES ON THIS PUBLICATION, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.
Although aicas GmbH has reviewed this publication, aicas GmbH MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESSED OR IMPLIED, WITH RESPECT TO THIS PUBLICATION, ITS QUALITY, ACCURACY, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS PUBLICATION IS PROVIDED AS IS, AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND ACCURACY.
IN NO EVENT WILL aicas GmbH BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT OR INACCURACY IN THIS PUBLICATION, even if advised of the possibility of such damages.
THE WARRANTIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESSED OR IMPLIED.

Contents

Preface **11**
Intended Audience of This Book 11
Contacting aicas 12
What is New in JamaicaVM 6.2 12
What is New in JamaicaVM 6.1 13
What is New in JamaicaVM 6.0 13

I Introduction **15**

1 Key Features of JamaicaVM **17**
1.1 Hard Realtime Execution Guarantees 17
1.2 Real-Time Specification for Java support 18
1.3 Minimal footprint 18
1.4 ROMable code 19
1.5 Native code support 19
1.6 Dynamic Linking 19
1.7 Supported Platforms 19
1.7.1 Development platforms 19
1.7.2 Target platforms 20
1.8 Fast Execution 21
1.9 Tools for Realtime and Embedded System Development 22

2 Getting Started **23**
2.1 Installation of JamaicaVM 23
2.1.1 Linux 24
2.1.2 Sun/Solaris 25
2.1.3 Windows 25
2.2 Installation of License Keys 26
2.2.1 Using the Standard Edition 26
2.2.2 Using the Personal Edition 27

2.3 JamaicaVM Directory Structure 28
2.4 Building and Running an Example Java Program 29
2.4.1 Host Platform 29
2.4.2 Target Platform 30
2.4.3 Improving Size and Performance 31
2.4.4 Overview of Further Examples 32
2.5 Notations and Conventions 32
2.5.1 Typographic Conventions 32
2.5.2 Argument Syntax 33
2.5.3 Jamaica Home and User Home 34

3 Tools Overview 35
3.1 Jamaica Java Compiler 35
3.2 Jamaica Virtual Machine 35
3.3 Jamaica Builder — Creating Target Executables 36
3.4 Jamaica ThreadMonitor — Monitoring Realtime Behavior 37

4 Support for the Eclipse IDE 39
4.1 Plug-in installation 39
4.1.1 Installation on Eclipse 39
4.1.2 Installation on Other IDEs 40
4.2 Setting up JamaicaVM Distributions 40
4.3 Setting Virtual Machine Parameters 41
4.4 Building applications with Jamaica Builder 41
4.4.1 Getting started 41
4.4.2 Jamaica Buildfiles 41

II Tools Usage and Guidelines 45

5 Performance Optimization 47
5.1 Creating a profile 47
5.1.1 Creating a profiling application 47
5.1.2 Using the profiling VM 49
5.1.3 Dumping a profile via network 49
5.1.4 Creating a micro profile 50
5.2 Using a profile with the Builder 50
5.2.1 Building with a profile 50
5.2.2 Building with multiple profiles 51
5.3 Interpreting the profiling output 52
5.3.1 Format of the profile file 52

5.3.2 Example 56

6 Reducing Footprint and Memory Usage 59
6.1 Compilation 59
6.1.1 Suppressing Compilation 59
6.1.2 Using Default Compilation 61
6.1.3 Using a Custom Profile 62
6.1.4 Code Optimization by the C Compiler 64
6.1.5 Full Compilation 65
6.2 Smart Linking 66
6.3 API Library Classes and Resources 68
6.4 RAM Usage 69
6.4.1 Measuring RAM Demand 69
6.4.2 Memory Required for Threads 71
6.4.3 Memory Required for Line Numbers 74

7 Memory Management Configuration 77
7.1 Configuration for soft-realtime applications 77
7.1.1 Initial heap size 77
7.1.2 Maximum heap size 78
7.1.3 Finalizer thread priority 78
7.1.4 Reserved memory 79
7.1.5 Using a GC thread 80
7.1.6 Stop-the-world Garbage Collection 80
7.1.7 Recommendations 81
7.2 Configuration for hard-realtime applications 82
7.2.1 Usage of the Memory Analyzer tool 82
7.2.2 Measuring an application's memory requirements 82
7.2.3 Fine tuning the final executable application 84
7.2.4 Constant Garbage Collection Work 86
7.2.5 Comparing dynamic mode and constant GC work mode . 87
7.2.6 Determination of the worst case execution time of an allocation 88
7.2.7 Examples 88

8 Debugging Support 91
8.1 Enabling the Debugger Agent 91
8.2 Configuring the IDE to Connect to Jamaica 92
8.3 Reference Information 92

9 The Real-Time Specification for Java **95**
9.1 Realtime programming with the RTSJ . . . 95
9.1.1 Thread Scheduling . . . 96
9.1.2 Memory Management . . . 96
9.1.3 Synchronization . . . 96
9.1.4 Example . . . 97
9.2 Realtime Garbage Collection . . . 98
9.3 Relaxations in JamaicaVM . . . 98
9.3.1 Use of Memory Areas . . . 99
9.3.2 Thread Priorities . . . 99
9.3.3 Runtime checks for NoHeapRealtimeThread . . . 99
9.3.4 Static Initializers . . . 100
9.3.5 Class PhysicalMemoryManager . . . 100
9.4 Strict RTSJ Semantics . . . 100
9.4.1 Use of Memory Areas . . . 100
9.4.2 Thread priorities . . . 101
9.4.3 Runtime checks for NoHeapRealtimeThread . . . 101
9.4.4 Static Initializers . . . 101
9.4.5 Class PhysicalMemoryManager . . . 102
9.5 Limitations of RTSJ Implementation . . . 102

10 Realtime Programming Guidelines **103**
10.1 General . . . 103
10.2 Computational Transparency . . . 103
10.2.1 Efficient Java Statements . . . 104
10.2.2 Non-Obvious Slightly Inefficient Constructs . . . 106
10.2.3 Statements Causing Implicit Memory Allocation . . . 107
10.2.4 Operations Causing Class Initialization . . . 109
10.2.5 Operations Causing Class Loading . . . 110
10.3 Supported Standards . . . 111
10.3.1 Real-Time Specification for Java . . . 111
10.3.2 Java Native Interface . . . 113
10.3.3 Java 2 Micro Edition . . . 114
10.4 Memory Management . . . 114
10.4.1 Memory Management of RTSJ . . . 115
10.4.2 Finalizers . . . 116
10.4.3 Configuring a Realtime Garbage Collector . . . 117
10.4.4 Programming with the RTSJ and Realtime Garbage Collection . . . 118
10.4.5 Memory Management Guidelines . . . 119
10.5 Scheduling and Synchronization . . . 120

10.5.1 Schedulable Entities . . . 120
10.5.2 Synchronization . . . 122
10.5.3 Scheduling Policy and Priorities . . . 125
10.6 Libraries . . . 127
10.7 Summary . . . 127
10.7.1 Efficiency . . . 127
10.7.2 Memory Allocation . . . 128
10.7.3 EventHandlers . . . 128
10.7.4 Monitors . . . 129

11 Multicore Guidelines 131
11.1 Tool Usage . . . 131
11.2 Setting Thread Affinities . . . 132
11.2.1 Communication through Shared Memory . . . 132
11.2.2 Performance Degradation on Locking . . . 133
11.2.3 Periodic Threads . . . 133
11.2.4 Rate-Monotonic Analysis . . . 134
11.2.5 The Operating System's Interrupt Handler . . . 134

III Tools Reference 135

12 The Jamaica Java Compiler 137
12.1 Usage of jamaicac . . . 137
12.1.1 Classpath options . . . 137
12.1.2 Compliance options . . . 138
12.1.3 Warning options . . . 138
12.1.4 Debug options . . . 139
12.1.5 Other options . . . 139
12.2 Environment Variables . . . 139

13 The Jamaica Virtual Machine Commands 141
13.1 jamaicavm — the Standard Virtual Machine . . . 141
13.1.1 Command Line Options . . . 142
13.1.2 Extended Command Line Options . . . 143
13.2 Variants of jamaicavm . . . 145
13.2.1 jamaicavm_slim . . . 145
13.2.2 jamaicavmm . . . 145
13.2.3 jamaicavmp . . . 146
13.2.4 jamaicavmdi . . . 147
13.3 Environment Variables . . . 147

13.4 Java Properties . . . 149
13.4.1 User-Definable Properties . . . 149
13.4.2 Predefined Properties . . . 153
13.5 Exitcodes . . . 154

14 The Jamaica Builder **157**
14.1 How the Builder tool works . . . 157
14.2 Builder Usage . . . 157
14.2.1 General . . . 160
14.2.2 Classes, files and paths . . . 161
14.2.3 Smart linking . . . 167
14.2.4 Profiling and compilation . . . 169
14.2.5 Heap and stack configuration . . . 171
14.2.6 Threads, priorities and scheduling . . . 173
14.2.7 Parallel Execution . . . 178
14.2.8 GC configuration . . . 178
14.2.9 RTSJ settings . . . 181
14.2.10 Native code . . . 184
14.3 Builder Extended Usage . . . 184
14.3.1 General . . . 184
14.3.2 Classes, files and paths . . . 185
14.3.3 Profiling and compilation . . . 186
14.3.4 Parallel Execution . . . 189
14.3.5 RTSJ settings . . . 190
14.3.6 Native code . . . 190
14.4 Environment Variables . . . 191
14.5 Exitcodes . . . 192

15 The Jamaica ThreadMonitor **193**
15.1 Run-time system configuration . . . 193
15.2 Control Window . . . 194
15.2.1 Control Window Menu . . . 195
15.3 Data Window . . . 197
15.3.1 Data Window Navigation . . . 198
15.3.2 Data Window Menu . . . 198
15.3.3 Data Window Context Window . . . 200
15.3.4 Data Window Tool Tips . . . 200
15.3.5 Worst-Case Execution Time Window . . . 200

16 Jamaica and the Java Native Interface (JNI) 203
16.1 Using JNI . . . 203
16.2 The Jamaicah Command . . . 206
16.2.1 General . . . 206
16.2.2 Classes, files, and paths . . . 207
16.2.3 Environment Variables . . . 208

17 Building with Apache Ant 209
17.1 Task Declaration . . . 209
17.2 Task Usage . . . 210
17.2.1 Jamaica Builder and Jamaicah . . . 210
17.2.2 C Compiler . . . 211
17.3 Setting Environment Variables . . . 212

IV Additional Information 213

A FAQ — Frequently Asked Questions 215
A.1 Software Development Environments . . . 215
A.2 JamaicaVM and Its Tools . . . 216
A.2.1 JamaicaVM . . . 216
A.2.2 JamaicaVM Builder . . . 217
A.2.3 Third Party Tools . . . 219
A.3 Supported Technologies . . . 219
A.3.1 Cryptography . . . 219
A.3.2 Fonts . . . 220
A.3.3 Serial Port . . . 221
A.3.4 Realtime Support and the RTSJ . . . 221
A.3.5 Remote Method Invocation (RMI) . . . 222
A.3.6 OSGi . . . 224
A.4 Target-Specific Issues . . . 224
A.4.1 VxWorks . . . 224

B Information for Specific Targets 227
B.1 Operating Systems . . . 227
B.1.1 VxWorks . . . 227
B.1.2 Windows . . . 235
B.1.3 WindowsCE . . . 236
B.1.4 OS-9 . . . 237
B.1.5 QNX . . . 237
B.2 Processor Architectures . . . 238

B.2.1 ARM 238

C Heap Usage for Java Datatypes **241**

D Limitations **243**
D.1 VM Limitations 243
D.2 Builder Limitations 245
D.3 Multicore Limitations 246

E Internal Environment Variables **247**

Preface

The Java programming language, with its clear syntax and semantics, is used widely for the creation of complex and reliable systems. Development and maintenance of these systems benefit greatly from object-oriented programming constructs such as dynamic binding and automatic memory management. Anyone who has experienced the benefits of these mechanisms on software development productivity and improved quality of resulting applications will find them essential when developing software for embedded and time-critical applications.

This manual describes JamaicaVM, a Java implementation that brings technologies that are required for embedded and time critical applications and that are not available in classic Java implementations. This enables this new application domain to profit from the advantages that have provided an enormous boost to most other software development areas.

Intended Audience of This Book

Most developers familiar with Java environments will quickly be able to use the tools provided with JamaicaVM to produce immediate results. It is therefore tempting to go ahead and develop your code without studying this manual further.

Even though immediate success can be achieved easily, we recommend that you have a closer look at this manual, since it provides a deeper understanding of how the different tools work and how to achieve the best results when optimizing for runtime performance, memory demand or development time.

The JamaicaVM tools provide a myriad of options and settings that have been collected in this manual. Developing a basic knowledge of what possibilities are available may help you to find the right option or setting when you need it. Our experience is that significant amounts of development time can be avoided by a good understanding of the tools. Learning about the correct use of the JamaicaVM tools is an investment that will quickly pay-off during daily use of these tools!

This manual has been written for the developer of software for embedded and time-critical applications using the Java programming language. A good under-

standing of the Java language is expected from the reader, while a certain familiarity with the specific problems that arise in embedded and realtime system development is also helpful.

This manual explains the use of the JamaicaVM tools and the specific features of the Jamaica realtime virtual machine. It is not a programming guidebook that explains the use of the standard libraries or extensions such as the Real-Time Specification for Java. Please refer to the JavaDoc documentation of these libraries provided with JamaicaVM (see § 2.3).

Contacting aicas

Please contact aicas or one of aicas's sales partners to obtain a copy of JamaicaVM for your specific hardware and RTOS requirements, or to discuss licensing questions for the Jamaica binaries or source code. The full contact information for the aicas main offices is reproduced in the front matter of this manual (page 2). The current list of sales partners is available online at `http://www.aicas.com/sales.html`.

An evaluation version of JamaicaVM may be downloaded from the aicas web site at `http://www.aicas.com/download.html`.

Please help us improve this manual and future versions of JamaicaVM. E-mail your bug reports and comments to `bugs@aicas.com`. Please include the exact version of JamaicaVM you use, the host and target systems you are developing for and all the information required to reproduce the problem you have encountered.

What is New in JamaicaVM 6.2

Version 6.2 of JamaicaVM introduces multicore support for several important platforms including Linux, QNX and VxWorks. Other new features include:

- 64-Bit support with a Java heap of up to 120GB (for Linux, other targets can be provided on request).
- An Ant Task for invoking the C compiler in a platform independent manner. This can help creating applications based on JNI.
- A reform to the handling of the global constant pool in JamaicaVM can lead to significant reductions in the size of the binaries created by JamaicaVM Builder.
- Improved handling of Unicode characters used in Java identifiers.

What is New in JamaicaVM 6.1

Version 6.1 of JamaicaVM provides several new features as well as performance and stability improvements.

- The JamaicaVM *Personal Edition* allows prolonged development using the JamaicaVM tools without the need for obtaining a development license. This is attractive for evaluations that take longer than the validity of a short-term evaluation license. The Personal Edition is limited to Linux x86 as development and target platform.
- Hot code replacement through the JVMTI functions PopFrame and RedefineClasses. This can increase productivity in Eclipse debug sessions substantially.
- Ant Tasks now provide support for passing environment variables to cross compilers called by JamaicaVM Builder.
- The JamaicaVM Builder provides an option for building specific cryptographic policy files into an application. Note that these files still need to be requested from aicas.

Performance improvements include handling of large objects and the standard library class `StringBuilder`.

What is New in JamaicaVM 6.0

Version 6.0 of JamaicaVM provides numerous enhancements. The most important additions are:

- Support for Java 6.0 features including Java 6.0 source and class file compatibility.
- The standard classes library is based on the OpenJDK code. This leads to a more conformant implementation of the standard classes, in particular AWT and Swing features.
- Extended range of supported graphics systems. New are:
  - DirectFB (Linux)
  - Windows
  - GF (QNX)

This adds to existing support for X11 (Linux), WindML (VxWorks), GDI (WindowsCE) and Maui (OS9).

- The Java 2D API, a set of classes for advanced 2D graphics and imaging, is supported on all graphics systems.
- Bindings for OpenGL, OpenGL-ES and OpenGL-SC are available.
- Improved handling of system resources such as files and sockets. Even though a Java application may fail to correctly close or release these resources, the VM now tracks these and releases them on shutdown.
- Full crypto support. By default, cryptographic strength is limited to 48 Bits; this is due to export restrictions. Please contact aicas for details.
- Dynamic loading of native libraries (on selected platforms only).

For user-relevant changes between minor releases of JamaicaVM, see the release notes, which are provided in the Jamaica installation, folder `doc`, file `RELEASE_NOTES`.

Part I

Introduction

Chapter 1

Key Features of JamaicaVM

The Jamaica Virtual Machine (JamaicaVM) is an implementation of the Java Virtual Machine Specification. It is a runtime system for the execution of applications written for the Java 6 Standard Edition. It has been designed for realtime and embedded systems and offers unparalleled support for this target domain. Among the extraordinary features of JamaicaVM are:

- Hard realtime execution guarantees
- Support for the Real-Time Specification for Java, Version 1.0.2
- Minimal footprint
- ROMable code
- Native code support
- Dynamic linking
- Supported platforms
- Fast execution
- Powerful tools for timing and performance analysis

1.1 Hard Realtime Execution Guarantees

JamaicaVM is the only implementation that provides hard realtime guarantees for all features of the languages together with high performance runtime efficiency. This includes dynamic memory management, which is performed by the JamaicaVM garbage collector.

All threads executed by the JamaicaVM are realtime threads, so there is no need to distinguish realtime from non-realtime threads. Any higher priority thread is guaranteed to be able to preempt lower priority threads within a fixed worst-case delay. There are no restrictions on the use of the Java language to program realtime code: Since the JamaicaVM executes all Java code with hard realtime guarantees, even realtime tasks can use the full Java language, i.e., allocate objects, call library functions, etc. No special care is needed. Short worst-case execution delays can be given for any code.

1.2 Real-Time Specification for Java support

JamaicaVM provides an industrial-strength implementation of the Real-Time Specification for Java Specification (RTSJ) V1.0.2 (see [2]) for a wide range of realtime operating systems available on the market. It combines the additional APIs provided by the RTSJ with the predictable execution obtained through realtime garbage collection and a realtime implementation of the virtual machine.

1.3 Minimal footprint

JamaicaVM itself occupies less than 1 MB of memory (depending on the target platform), such that small applications that make limited use of the standard libraries typically fit into a few MB of memory. The biggest part of the memory required to store a Java application is typically the space needed for the application's class files and related resources such as character encodings. Several measures are taken by JamaicaVM to minimize the memory needed for Java classes:

- Compaction: Classes are represented in an efficient and compact format to reduce the overall size of the application.
- Smart Linking: JamaicaVM analyzes the Java applications to detect and remove any code and data that cannot be accessed at run-time.
- Fine-grained control over resources such as character encodings, time zones, locales, supported protocols, etc.

Compaction typically reduces the size of class file data by over 50%, while smart linking allows for much higher gains even for non-trivial applications.

This footprint reduction mechanism allows the usage of complex Java library code, without worrying about the additional memory overhead: Only code that is really needed by the application is included and is represented in a very compact format.

1.4 ROMable code

The JamaicaVM allows class files to be linked with the virtual machine code into a standalone executable. The resulting executable can be stored in ROM or flash-memory since all files required by a Java application are packed into the standalone executable. There is no need for file-system support on the target platform, as all data required for execution is contained in the executable application.

1.5 Native code support

The JamaicaVM implements the Java Native Interface V1.2 (JNI). This allows for direct embedding of existing native code into Java applications, or to encode hardware-accesses and performance-critical code sections in C or machine code routines. The usage of the Java Native Interface provides execution security even with the presence of native code, while binary compatibility with other Java implementations is ensured. Unlike other Java implementations, JamaicaVM provides exact garbage collection even with the presence of native code. Realtime guarantees for the Java code are not affected by the presence of native code.

1.6 Dynamic Linking

One of the most important features of Java is the ability to dynamically load code in the form of class files during execution, e.g., from a local file system or from a remote server. The JamaicaVM supports this dynamic class loading, enabling the full power of dynamically loaded software components. This allows, for example, on-the-fly reconfiguration, hot swapping of code, dynamic additions of new features, or applet execution.

1.7 Supported Platforms

During development special care has been taken to reduce porting effort of the JamaicaVM to a minimum. JamaicaVM is implemented in C using the GNU C compiler. Threads are based on native threads of the operating system.[1]

1.7.1 Development platforms

Jamaica is available for the following development platforms (host systems):

[1] POSIX threads under many Unix systems.

- Linux
- SunOS/Solaris
- Windows

1.7.2 Target platforms

With JamaicaVM, application programs for a large number of platforms (target systems) can be built. The operating systems listed in this section are supported as target systems only. You may choose any other supported platform as a development environment on which the Jamaica Builder runs to generate code for the target system.

1.7.2.1 Realtime Operating Systems

- INTEGRITY (on request)
- Linux/RT
- OS-9 (on request)
- PikeOS
- QNX
- RTEMS (on request)
- ThreadX (on request)
- WinCE
- VxWorks

1.7.2.2 Non-Realtime Operating Systems

Applications built with Jamaica on non-realtime operating systems may be interrupted non-deterministically by other threads of the operating systems. However, Jamaica applications are still deterministic and there are still no unexpected interrupts within Jamaica application themselves, unlike with standard Java Virtual Machines.

- Linux
- SunOS/Solaris
- Windows

1.7.2.3 Processor Architectures

JamaicaVM is highly processor architecture independent. New architectures can be supported easily. Currently, Jamaica runs on the following processor architectures:

- ARM (StrongARM, XScale, . . .)
- ERC32 (on request)
- MIPS (on request)
- Nios
- PowerPC
- SH-4 (on request)
- Sparc
- x86

Ports to any required combination of target OS and target processor can be supported with little effort. Clear separation of platform-dependent from platform-independent code reduces the required porting effort for new target OS and target processors. If you are interested in using Jamaica on a specific target OS and target processor combination or on any operating system or processor that is not listed here, please contact aicas .

1.8 Fast Execution

The JamaicaVM interpreter performs several selected optimizations to ensure optimal performance of the executed Java code. Nevertheless, realtime and embedded systems are often very performance-critical as well, so a purely interpreted solution may be unacceptable. Current implementations of Java runtime-systems use just-in-time compilation technologies that are not applicable in realtime systems: The initial compilation delay breaks all realtime constraints.

The Jamaica compilation technology attacks the performance issue in a new way: methods and classes can selectively be compiled as a part of the build process (static compilation). C-code is used as an intermediary target code, allowing easy porting to different target platforms. The Jamaica compiler is tightly integrated into the memory management system, allowing highest performance and reliable realtime behavior. No conservative reference detection code is required, enabling fully exact and predictable garbage collection.

1.9 Tools for Realtime and Embedded System Development

JamaicaVM comes with a set of tools that support the development of applications for realtime and embedded systems

- Jamaica Builder: a tool for creating a single executable image out of the Jamaica Virtual Machine and a set of Java classes. This image can be loaded into flash-memory or ROM, avoiding the need for a file-system in the target platform.

  For most effective memory usage, the Jamaica Builder finds the amount of memory that is actually used by an application. This allows both system memory and heap size to be precisely chosen for optimal run-time performance. In addition, the Builder enables the detection of performance critical code to control the static compiler for optimal results.

- Thread Monitor: enables to analyze and fine-tune the behavior of threaded Java applications.[2]

- VeriFlux: a static analysis tool for the object-oriented domain that enables to prove the absence of potential faults such as null pointer exceptions or deadlocks in Java programs.[2]

[2]Thread Monitor and VeriFlux are not part of the standard Jamaica license.

Chapter 2

Getting Started

2.1 Installation of JamaicaVM

A release of the JamaicaVM tools consists of an info file with detailed information about the host and target platform and optional features such as graphics support, and a package for the Jamaica binaries, library and documentation files. The Jamaica version, build number, host and target platform and other properties of a release is encoded as *release identification string* in the names of info and package file according to the following scheme:

`Jamaica-`*version*`-`*build*`[-`*features*`]-`*host*`[-`*target*`].info`
`Jamaica-`*version*`-`*build*`[-`*features*`]-`*host*`[-`*target*`].`*suffix*

Package files with the following package suffixes are released.

Host Platform	Suffix	Package Kind
Linux	`rpm`	Package for the `rpm` package manager
	`tar.gz`	Compressed tape archive file
Windows	`exe`	Interactive installer
	`zip`	Windows zip file
Solaris	`tar.gz`	Compressed tape archive file

In order to install the JamaicaVM tools, the following steps are required:

- Unpack and install the Jamaica binaries, library and documentation files,
- Configure the tools for host and target platform (C compiler and native libraries),
- Set environment variables.
- Install license keys.

The actual installation procedure varies from host platform to host platform; see the sections below. For additional, target specific installation hints please check § B.

2.1.1 Linux

2.1.1.1 Unpack and Install Files

The default is a system-wide installation of Jamaica. Super user privileges are required. If the `rpm` package manager is available, this is the recommended method:

```
> rpm -i Jamaica-release-identification-string.rpm
```

Otherwise, unpack the compressed tape archive file and run the installation script as follows:

```
> tar xfz Jamaica-release-identification-string.tar.gz
> ./Jamaica.install
```

Both methods will install the Jamaica tools in the following directory, which is referred to as *jamaica-home*:

```
/usr/local/jamaica-version-build
```

In addition, the symbolic link `/usr/local/jamaica` is created, which points to *jamaica-home*, and symbolic links to the Jamaica executables are created in `/usr/bin`, so it is not necessary to extend the `PATH` environment variable.

In order to uninstall the Jamaica tools, depending on the used installation method, either use the `erase` option of `rpm` or the provided uninstall script `Jamaica.remove`.

If super user privileges are not available, the tools may alternatively be installed locally in a user's home directory:

```
> tar xfz Jamaica-release-identification-string.tar.gz
> tar xf Jamaica.ss
```

This will install the Jamaica tools in `usr/local/jamaica-version-build` relative to the current working directory. Symbolic links to the executables are created in `usr/bin`, so they will not be on the default path for executables.

2.1.1.2 Configure Platform-Specific Tools

In order for the Jamaica Builder to work, platform-specific tools such as the C compiler and linker and the locations of the libraries (SDK) need to be specified. This is done by editing the appropriate configuration file, named `jamaica.conf`, for the target (and possibly also the host).

The precise location of the configuration file depends on the platform:

jamaica-home/target/*platform*/etc/jamaica.conf

For the full Jamaica directory structure, please refer to § 2.3. Note that the configuration for the host platform is also located in a target directory.

The following properties need to be set appropriately in the configuration files:

Property	Value
Xcc	C compiler executable
Xld	Linker executable
Xstrip	Strip utility executable
Xinclude	Include path
XlibraryPaths	Library path

Environment variables may be accessed in the configuration files through the notation ${*VARIABLE*}. For executables that are on the standard search path (environment variable PATH), it is sufficient to give the name of the executable.

2.1.1.3 Set Environment Variables

The environment variable JAMAICA must be set to *jamaica-home*. It is recommended to also add *jamaica-home*/bin to the system path. On bash:

```
> export JAMAICA=jamaica-home
> export PATH=jamaica-home/bin:$PATH
```

On csh:

```
> setenv JAMAICA jamaica-home
> setenv PATH jamaica-home/bin:$PATH
```

2.1.2 Sun/Solaris

The release for Solaris is provided as compressed tape archive. Please follow the installation instructions in § 2.1.1.

2.1.3 Windows

On Windows the recommended means of installation is using the interactive installer, which may be launched by double-clicking the file

Jamaica-*release-identification-string*.exe

in the Explorer, or by executing it in the `CMD` shell. You will be asked to provide a destination directory for the installation and the locations of tools and SDK for host and target platforms. The destination directory is referred to as *jamaica-home*. It defaults to the subdirectory `jamaica` in Window's default program directory — for example, `C:\Programs\jamaica`, if an English language locale is used. Defaults for tools and SDKs are obtained from the registry. The installer will set the environment variable `JAMAICA` to *jamaica-home*.

An alternative installation method is to unpack the Windows zip file into a suitable installation destination directory. For configuration of platform-specific tools, follow the instructions provided in § 2.1.1. In order to set the `JAMAICA` environment variable to *jamaica-home*, open the Control Panel, choose System, select Advanced System Settings,[1] choose the tab Advanced and press Environment Variables. It is also recommended to add *jamaica-home*`\bin` to the `PATH` environment variable in order to be able to run the Jamaica executables conveniently.

2.2 Installation of License Keys

There are two different editions of JamaicaVM, a *Standard Edition* and a *Personal Edition*, that support different licensing models. The Standard Edition requires license keys for using the tools. License keys are provided with support contracts or for evaluation of JamaicaVM. The Personal Edition requires an online key for using the tools, and for running the VMs and application executables built with JamaicaVM tools. It does not require a support contract, but it requires an internet connection for checking keys online. The Personal Edition is intended for prolonged evaluations of JamaicaVM. It is available for selected host and target platforms only.

2.2.1 Using the Standard Edition

In order to use the Standard Edition of JamaicaVM tools valid licenses are required. Evaluation keys are available with evaluation versions of JamaicaVM. License keys are provided in *key ring* files, which have the suffix `.aicas_key`. Prior to use, keys need to be installed. This is done with the aicas key installer utility `aicasKeyInstaller`, which is located in *jamaica-home*`/bin`. Simply execute the utility providing the key ring as command line argument:

```
> cd jamaica-home/bin
> ./aicasKeyInstaller jamaica.aicas_key
```

[1] Some Windows versions only.

This will extract the keys contained in `jamaica.aicas_key` and add the individual key files to *user-home*`/.jamaica`. Keys that are already installed are not overwritten. The utility reports which keys get installed and which tools they enable. Installed keys are for individual tools. Of the tools documented in this manual, the Builder (see § 14) and the Thread Monitor (see § 15) require keys. [2]

2.2.2 Using the Personal Edition

To run any commands of the JamaicaVM Personal Edition or built with the builder tool of the Personal Edition, an online key is required. The online key can be requested at `http://signup.aicas.com`, it will be sent to you by e-mail.

Along with the JamaicaVM Personal Edition the aicas License Provider utility `aicasLicenseProvider` is provided as a separate download. This program performs the license checking, it communicates with the JamaicaVM commands or built applications running on the same machine and with aicas' servers to request permissions to run. If required, the `aicasLicenseProvider` will open user dialogs to request input such as the online key that was emailed to you, and to confirm that you give permission to transfer data to aicas' servers. No data will be transferred unless you confirm the corresponding dialog.

Before you can use any of the tools of the JamaicaVM Personal Edition, the `aicasLicenseProvider` must be started first:

```
> cd jamaica-home/bin
> ./aicasLicenseProvider
```

This program needs to run while the JamaicaVM tools are used, so it should be started in a separate shell or sent to the background. It can be invoked in non-interactive mode if pop-up dialogs are not desired or no graphics system is available:

```
> ./aicasLicenseProvider -nonInteractive -key online key
```

Please be aware that in non-interactive mode a hashcode of the Java main class, the user and host name and the MAC address of the system will be transferred without confirmation.

To find out more about the `aicasLicenseProvider` command, use the `-help` option.

! License checking requires a direct connection to servers at aicas. Communication via proxies is not supported.

[2] For old versions of JamaicaVM (before Version 6.0, Release 3), the key installer is provided separately from the distribution package. For old versions of the installer, the key installer and the key ring must be placed into the same directory.

jamaica-home	
`+- bin`	Host tool chain executables
`+- doc`	
`  +- build.info`	Comprehensive Jamaica distribution information
`  +- jamaicavm_manual.pdf`	
`  |`	Jamaica tool chain user manual (this manual)
`  +- jamaica_api`	Jamaica API specification (Javadoc)
`  +- README-*.txt`	Host platform specific documentation starting points
`  +- RELEASE_NOTES`	User-relevant changes in the present release
`  +- UNSUPPORTED`	Unsupported features list
`  +- *.1`	Tool documentation in Unix man page format
`+- etc`	Host platform configuration files
`+- lib`	Libraries for the development tools
`+- license`	aicas evaluation license, third party licenses
`+- target`	
`  +- linux-x86`	Target specific files for the target `linux-x86`
`    +- bin`	Virtual machine executables (some platforms only)
`    +- etc`	Default target platform configuration files
`    +- examples`	Example applications
`    +- include`	System JNI header files
`    +- lib`	Development and runtime libraries, resources
`    +- prof`	Default profiles

Table 2.1: JamaicaVM Directory Structure

2.3 JamaicaVM Directory Structure

The Jamaica installation directory is called *jamaica-home*. The environment variable `JAMAICA` should be set to this path (see the installation instructions above). After successful installation, the following directory structure as shown in Tab. 2.1 is created (in this example for a Linux x86 system).

The Jamaica API specification may be browsed with an ordinary web browser. Its format is compatible with common IDEs such as Eclipse and Netbeans. If the Jamaica Eclipse Plug-In is used (see § 4), Eclipse will automatically use the API specification of the selected Jamaica runtime environment. The Real-Time Specification for Java is part of the standard Jamaica API.

The number of target systems supported by a distribution varies. The `target` directory contains an entry for each supported target platform. Typically, a Jamaica distribution provides support for the target platform that hosts the tool chain, as well as for an embedded or real-time operating system.

2.4 Building and Running an Example Java Program

A number of sample applications is provided. These are located in the directory *jamaica-home*/target/*platform*/examples. In the following instructions it is assumed that a Unix host system is used. For Windows, please note that the Unix path separator character "/" should be replaced by \".

Before using the examples, it is recommended to copy them from the installation directory to a working location — that is, copy each of the directories *jamaica-home*/*platform*/examples to *user-home*/examples/*platform*.

The HelloWorld example is an excellent starting point for getting acquainted with the JamaicaVM tools. In this section, the main tools are used to build an application executable for a simple HelloWorld both for the host and target platforms. First, the command-line tools are used. Later we switch to using ant build files.

Below, it is assumed that the example directories have been copied to *user-home*/examples/*host* and *user-home*/examples/*target* for host and target platforms respectively.

2.4.1 Host Platform

In order to build and run the HelloWorld example on the host platform, go to the corresponding examples directory:

```
> cd user-home/examples/host
```

Depending on your host platform, *host* will be linux-x86_64 (in rare cases linux-x86), windows-x86 or solaris-sparc.

First, the Java source code needs to be compiled to byte code. This is done with jamaicac, Jamaica's version of javac. The source code resides in the src folder, and we wish to generate byte code in a classes folder, which must be created if not already present:

```
> mkdir classes
> jamaicac -d classes src/HelloWorld.java
```

Before generating an executable, we test the byte code with the Jamaica virtual machine:

```
> jamaicavm -cp classes HelloWorld
              Hello        World!
           Hello       World!
        Hello     World!
     Hello    World!
   Hello   World!
```

```
 Hello  World!
 [...]
```

Having convinced ourselves that the program exhibits the desired behavior, we now generate an executable with the Jamaica Builder. In the context of the JamaicaVM Tools, one speaks of *building* an application.

```
> jamaicabuilder -cp classes -interpret HelloWorld
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/HelloWorld__.c
 + tmp/HelloWorld__.h
 * C compiling 'tmp/HelloWorld__.c'
 + tmp/HelloWorld__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                        initial               max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                256MB
GC data:                128KB                 16MB
TOTAL:                 3328KB                343MB
```

The builder has now generated the executable `HelloWorld`.

```
> ./HelloWorld
              Hello        World!
           Hello       World!
        Hello     World!
     Hello    World!
   Hello   World!
 Hello  World!
 [...]
```

2.4.2 Target Platform

With the JamaicaVM Tools, building an application for the target platform is as simple as for the host platform. First go to the corresponding examples directory:

```
> cd user-home/examples/platform
```

Then compile and build the application specifying the target platform.

```
> mkdir classes
> jamaicac -useTarget platform -d classes src/HelloWorld.java
> jamaicabuilder -target=platform -cp=classes -interpret HelloWorld
```

The target specific binary `HelloWorld` is generated. For instructions how to launch this on the target operating system, please consult the documentation of the operating system. Additional target-specific hints are provided in § B.

2.4.3 Improving Size and Performance

The application binaries in the previous two sections provide decent size optimization but no performance optimization at all. The JamaicaVM Tools offer a wide range of controls to fine tune the size and performance of a built application. These optimizations are mostly controlled through command line options of the Jamaica Builder.

Sets of optimizations for both speed and application size are provided with the HelloWorld example in an `ant` buildfile (`build.xml`). In order to use the buildfile, type `ant` *build-target* where *build-target* is one of the build targets of the example. For example,

```
> ant HelloWorld
```

will build the unoptimized HelloWorld example. If building for a Windows platform, the command

```
> ant HelloWorld.exe
```

should be used. In order to optimize for speed, use the build target `HelloWorld_profiled`, in order to optimize for application size, use `HelloWorld_micro`. The following is the list of all build targets available for the HelloWorld example; the optional `.exe` suffix is for Windows platforms only:

`HelloWorld[.exe]` Build an application in interpreted mode. The generated binary is `HelloWorld`.

`HelloWorld_profiled[.exe]` Build a statically compiled application based on a profile run. The generated binary is `HelloWorld_profiled`.

`HelloWorld_micro[.exe]` Build an application with optimized memory demand. The generated binary is `HelloWorld_micro`.

`classes` Convert Java source code to byte code.

`all` Build all three applications.

`run` Run all three applications — only useful on the host platform.

`clean` Remove all generated files.

Example	Demonstrates	Platforms
`HelloWorld`	Basic Java	all
`RTHelloWorld`	Real-time threads (RTSJ)	all
`SwingHelloWorld`	Swing graphics	with graphics
`caffeine`	CaffeineMark (tm) benchmark	all
`test_jni`	Java Native Interface	all
`net`	Network and internet	with network
`rmi`	Remote method invocation	with network
`DynamicLibraries`	Linking native code at runtime	selected platforms
`Queens`	Parallel execution	multicore systems

Table 2.2: Example applications provided in the target directories

2.4.4 Overview of Further Examples

For an overview of the available examples, see Tab. 2.2. Examples that require graphics or network support are only provided for platforms that support graphics or network, respectively. Each example comes with a `README` file that provides further information and lists the available build targets.

2.5 Notations and Conventions

Notations and typographic conventions used in this manual and by the JamaicaVM Tools in general are explained in the following sections.

2.5.1 Typographic Conventions

Throughout this manual, names of commands, options, classes, files etc. are set in this `monospaced` font. Output in terminal sessions is reproduced in *`slanted`* monospaced in order to distinguish it from user input. Entities in command lines and other user inputs that have to be replaced by suitable user input are shown in *italics*.

As little example, here is the description of the the Unix command-line tool `cat`, which outputs the content of a file on the terminal:

> Use `cat` *file* to print the content of *file* on the terminal. For example, the content of the file `song.txt` may be inspected thus:
>
> ```
> > cat song.txt
> Mary had a little lamb,
> ```

```
Little lamb, little lamb,
Mary had a little lamb,
Its fleece was white as snow.
```

In situations where suitable fonts are not available — say, in terminal output — entities to be replaced by the user are displayed in angular brackets. For example, `cat <file>` instead of `cat` *file*.

2.5.2 Argument Syntax

In the specification of command line arguments and options, the following notations are used.

Alternative: the pipe symbol "`|`" denotes alternatives. For example,

`-XobjectFormat=default|C|ELF`

means that the `XobjectFormat` option must be set to exactly one of the specified values `default`, `C` or `ELF`.

Option: optional arguments that may appear at most once are enclosed in brackets. For example,

`-heapSize=`*n*`[K|M]`

means that the `heapSize` option must be set to a (numeric) value *n*, which may be followed by either `K` or `M`.

Repetition: optional arguments that may be repeated are enclosed in braces. For example,

`-priMap=`*jp*`=`*sp*`{,`*jp*`=`*sp*`}`

means that the `priMap` accepts one or several comma-separated arguments of the form *jp*=*sp*. These are assignments of Java priorities to system priorities.

Alternative option names are indicated in parentheses. For example,

`-help(--help, -h, -?)`

means that the option `help` may be invoked by any one of `-help`, `--help`, `-h` and `-?`.

2.5.3 Jamaica Home and User Home

The file system location where the JamaicaVM Tools are installed is referred to as *jamaica-home*. In order for the tools to work correctly, the environment variable `JAMAICA` must be set to *jamaica-home* (see § 2.1).

The JamaicaVM Tools store user-related information such as license keys in the folder `.jamaica` inside the user's home directory. The user's home directory is referred to as *user-home*. On Unix systems it is usually `/home/`*user*, on Windows `C:\Users\`*user*. On Windows it may also reside inside the Documents and Settings folder — for example, `C:\Documents and Settings\`*user*.

Chapter 3

Tools Overview

The JamaicaVM tool chain provides all the tools required to process Java source code into an executable format on the target system. Fig. 3.1 provides an overview over this tool chain.

3.1 Jamaica Java Compiler

JamaicaVM uses Java source code files (see the Java Language Specification [4]) as input to first create platform independent Java class files (see the Java Virtual Machine Specification [7]) in the same way classical Java implementations do. JamaicaVM provides its own Java bytecode compiler, `jamaicac`, to do this translation. For a more detailed description of `jamaicac` see § 12.

We recommend using `jamaicac`. However, it is also possible to use your favorite Java source to bytecode compiler, including JDK's `javac` command, as long as you ensure that the bootclasspath is set properly to the Jamaica boot classes. These are located in the following JAR file:

jamaica-home`/target/`*platform*`/lib/rt.jar`

In addition, please note that JamaicaVM 6.2 uses Java 6 compatible class files and requires a Java compiler capable of interpreting Java 6 compatible class files.

3.2 Jamaica Virtual Machine

The command `jamaicavm` provides a version of the Jamaica virtual machine. It can be used directly to quickly execute a Java application. It is the equivalent to the `java` command that is used to run Java applications with SUN's JDK. A more detailed description of the `jamaicavm` and similar commands that are part of Jamaica will be given in § 13.

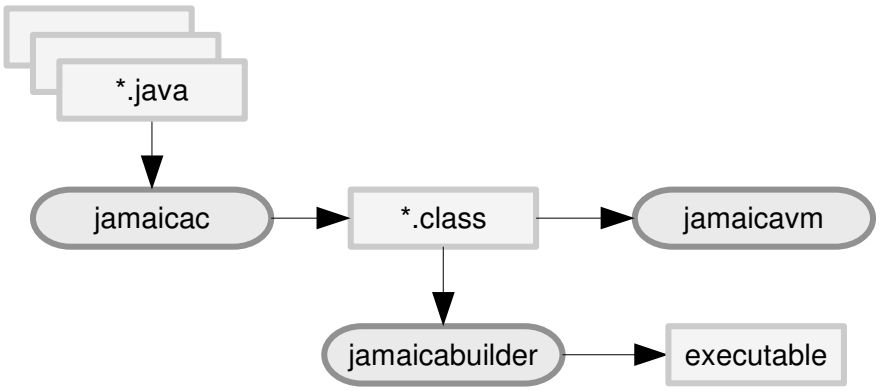


Figure 3.1: The Jamaica Toolchain

The `jamaicavm` first loads all class files that are required to start the application. It contains the Jamaica Java interpreter, which then executes the byte code commands found in these class files. Any new class that is referenced by a byte code instruction that is executed will be loaded on demand to execute the full application.

Applications running using the `jamaicavm` command are not very well optimized. There is no compiler that speeds up execution and no specific measures to reduce footprint are taken. We therefore recommend using the Jamaica builder presented in the next section and discussed in detail in § 14 to run Java applications with JamaicaVM on an embedded system.

3.3 Jamaica Builder — Creating Target Executables

In contrast to the `jamaicavm` command, `jamaicabuilder` does not execute the Java application directly. Instead, the builder loads all the classes that are part of a Java application and packages them together with the Jamaica runtime system (Java interpreter, class loader, realtime garbage collector, JNI native interface, etc.) into a stand-alone executable. This stand-alone executable can then be executed on the target system without needing to load the classes from a file system as is done by the `jamaicavm` command, but can instead directly proceed executing the byte codes of the application's classes that were built into the stand-alone executable.

The builder has the opportunity to perform optimizations on the Java application before it is built into a stand-alone executable. These optimizations reduce the memory demand (smart linking, bytecode compaction, etc.) and increase its

runtime performance (bytecode optimizations, profile-guided compilation, etc.). Also, the builder permits fine-grained control over the resources available to the application such as number of threads, heap size, stack sizes and enables the user to deactivate expensive functions such as dynamic heap enlargement or thread creation at runtime. A more detailed description of the builder is given in § 14.

3.4 Jamaica ThreadMonitor — Monitoring Realtime Behavior

The JamaicaVM ThreadMonitor enables to monitor the realtime behavior of applications and helps developers to fine-tune the threaded Java applications running on Jamaica run-time systems. These run-time systems can be either the Jamaica VM or any application that was created using the Jamaica Builder.

Chapter 4

Support for the Eclipse IDE

Integrated development environments (IDEs) make a software engineer's life easier by aggregating all important tools under one user interface. aicas provides a plug-in to integrate the JamaicaVM Virtual Machine and the JamaicaVM Builder into the Eclipse IDE, which is a popular IDE for Java.

4.1 Plug-in installation

The JamaicaVM plug-in can be installed and updated through the Eclipse plug-in manager.

4.1.1 Installation on Eclipse

The plug-in requires Eclipse 3.2 or later and a Java 1.5 compatible virtual machine. However, using the latest available Eclipse version and an up-to-date virtual machine is recommended. The following instructions refer to Eclipse 3.5. Earlier versions of Eclipse differ slightly in the menu item labels.

To install the plug-in, select the menu item

```
Help > Install New Software...,
```

add the aicas Update Site, whose location is `http://aicas.com/download/eclipse-plugin`, and install `JamaicaVM Tools`.[1] The plug-in is available after a restart of Eclipse.

To perform an update, select `Help > Check for updates...`. You will be notified of updates.

[1] Some web browsers may be unable to display the update site.

4.1.2 Installation on Other IDEs

The plug-in may also be used on development environments that are based on Eclipse such as WindRiver's WorkBench or QNX Momentics. These environments are normally not set up for Java development and may lack the Java Development Tools (JDT). In order to install these

- Identify the Eclipse version the development environment is derived from. This information is usually available in the `Help > About` dialog — for example, Eclipse 3.5.

- Some IDEs have the menu item for installing new software disabled by default. To enable it switch to the Resource Perspective: select `Window > Open Perspective > Other...` and choose `Resource`.

- Add the corresponding Eclipse Update Site, which is `http://download.eclipse.org/eclipse/updates/3.5` in this example, and install the JDT: select `Help > Install New Software...` and add the update site. Then uncheck "Group items by category" and select the package "Eclipse Java Development Tools". Installation may require to run the IDE in admin mode.

Restart the development environment before installing the JamaicaVM plug-in.

4.2 Setting up JamaicaVM Distributions

A Jamaica distribution must be made known to Eclipse and the Jamaica plug-in before it can be used. This is done by installing it as a Java Runtime Environment (JRE). In the global preferences dialog (usually `Window > Preferences`), open Section `Java > Installed JREs`, click `Add...`, select `JamaicaVM` and choose the Jamaica installation directory as the JRE home. The wizard will automatically provide defaults for the remaining fields.

After setting up a Jamaica distribution as a JRE, it can be used like any other JRE in Eclipse. For example, it is possible to choose Jamaica as a project specific environment for a Java project, either in the `Create Java Project` wizard, or by changing `JRE System Library` in the properties of an existing project. It is also possible to choose a Jamaica as default JRE for the workspace.

If you added a new Jamaica distribution and its associated JRE installation is not visible afterwards, please restart Eclipse.

4.3 Setting Virtual Machine Parameters

The JamaicaVM Virtual Machine is configured through runtime parameters, which — for example — control the heap size or the size of memory areas such as scoped memory. These settings are controlled via environment variables (refer to section § 13.3 for a list of available variables). To do so, create or open a run configuration of type `Java Application` in your project. The environment variables can be defined on the tab named `Environment`.

4.4 Building applications with Jamaica Builder

The plug-in extends Eclipse with support for the Jamaica Builder tool. In the context of this tool, the term "build" is used to describe the process of translating compiled Java class files into an executable file. Please note that in Eclipse's terminology, "build" means compiling Java source files into class files.

4.4.1 Getting started

In order to build your application with Jamaica Builder, you must create a Jamaica Buildfile. A wizard is available for creating a build file for an existing project with sources (the wizard needs to know the main class).

To use the wizard, invoke Eclipse's "New" dialog by choosing `File > New > Other...`, navigate to `Jamaica > Jamaica Buildfile`. Select a Jamaica distribution and target platform, choose a project in the workspace and specify the application's main class name.

After finishing the wizard, the newly created buildfile is opened in a graphical editor containing an overview page, a configuration page and a source page. You can review and modify the Jamaica Builder configuration on the second page, or in order to start the build process, click `Invoke Ant on this target` on the `Overview` page.

4.4.2 Jamaica Buildfiles

This section gives a more detailed introduction to Jamaica Buildfiles and the graphical editor to edit them easily.

4.4.2.1 Concepts

Jamaica Buildfiles are build files understood by Apache Ant. (See `http://ant.apache.org`.) These build files mainly consist of *targets* containing a

sequence of *tasks* which accomplish a functionality like compiling a set of Java classes. Many tasks come included with Ant, but tasks may also be provided by a third party. Third party tasks must be defined within the buildfile by a task definition (*taskdef*). Ant tasks that invoke the Jamaica Builder and other tools are part of the JamaicaVM tools. See § 17 for the available Ant tasks and further details on the structure of the Jamaica Buildfiles.

The Jamaica-specific tasks can be parameterized similarly to the tools they represent. We define the usage of such a task along with a set of options as a *configuration*.

We use the term Jamaica Buildfile to describe an Ant buildfile that defines at least one of the Jamaica-specific Ant tasks and contains one or many configurations.

The benefit of this approach is that configurations can easily be used outside of Eclipse, integrated in a build process and exchanged or stored in a version control system.

4.4.2.2 Using the editor

The editor for Jamaica Buildfiles consists of three or more pages. The first page is the `Overview` page. On this page, you can manage your configurations, task definitions and Ant properties. More information on this can be found in the following paragraphs. The pages after the `Overview` page represent a configuration. The last page displays the XML source code of the buildfile. Normally, you should not need to edit the source directly.

4.4.2.3 Configure Builder options

A configuration page consists of a header section and a body part. Using the controls in the header, you can request the build of the current configuration, change the task definition used by the configuration or add options to the body part. Each option in the configuration is displayed by an input mask, allowing you to perform various actions:

- Modify options. The input masks reflect the characteristics of their associated option, e.g. an option that expects a list will be displayed as a list control. Input masks that consists only of a text field show an asterisk (*) after the option name when modified. Please press `[Enter]` to accept the new value.

- Remove options. Each input mask has a `[x]` control that will remove the option from the configuration.

- Disable options. Options can also be disabled instead of removed, e.g. in order to test the configuration without a specific option. Uncheck the checkbox in front of an input mask to disable that option.

- Show help texts and load default values. The arrow control in the upper right corner brings up a context menu. You can show the option's description or load its default value (not available for all options).

The values of all options are immediately validated. If a value is not valid for a specific option, the input mask will be annotated with the text `invalid` and an error message is shown. Invalid options will be ignored when you try to build your application.

4.4.2.4 Multiple configurations

It is possible to store more than one configuration in a buildfile. Click `Add a Jamaica Builder target` to create a new minimal configuration. The new configuration will be displayed in a new page in the editor. A configuration can be removed on the `Overview` page by clicking `remove` after the configuration's name.

4.4.2.5 Multiple distributions

The plug-in uses task definitions (*taskdefs* in Ant's terminology) to link the configurations in a buildfile to a Jamaica distribution. Each Jamaica buildfile needs at least one of these taskdefs, however you can setup more to be used within the buildfile.

The section `Configured Jamaica tasks` shows the known task definitions and lets you add new or remove existing ones.

Task definitions can be *unbound*, meaning that the definition references a Jamaica distribution that is currently not available or not yet known to the plug-in. In such a case, you can create a new taskdef with the same name as the unbound one.

4.4.2.6 Ant properties

Ant properties provide a text-replacement mechanism within Ant buildfiles. The editor supports Ant properties[2] in option values. This is especially useful in conjunction with multiple configurations in one buildfile, when you create Ant properties for option values that are common to all configurations.

[2]The property task's `env` attribute is not supported.

4.4.2.7 Launch built application

The editor provides a simple way to launch the built application when it has been built for the host platform. If the wizard did not already generate a target named with a "run" prefix, click `Create new launch application target` to add a target that executes the binary that resulted from the specific Builder configuration.

Click `Invoke Ant on this target` to start the application. If the application needs runtime parameters, those can be added by clicking `(configure)` at the end of the arguments line.

Part II

Tools Usage and Guidelines

Chapter 5

Performance Optimization

The most fundamental measure employed by the Jamaica Builder to improve the performance of an application is to statically compile those parts that contribute most to the overall runtime. These parts are identified in a *profile run* of the application. Identifying these parts is called *profiling*. The profiling information is used by the Builder to decide which parts of an application need to be compiled and whether further optimizations such as inlining the code are necessary.

5.1 Creating a profile

The builder's `-profile` option and the `jamaicavmp` command provide simple means to profile an application. Setting the `-profile` option enables profiling. The builder will then link the application with the profiling version of the JamaicaVM libraries.

During profiling the Jamaica Virtual Machine counts, among other things, the number of bytecode instructions executed within every method of the application. The number of instructions can be used as a measure for the time spent in each method.

At the end of execution, the total number of bytecode instructions executed by each method is written to a file with the name of the main class of the Java application and the suffix `.prof`, such that it can be used for further processing. 'Hot spots' (the most likely sources for further performance enhancements by optimization) in the application can easily be determined using this file.

5.1.1 Creating a profiling application

The compilation technology of Jamaica's builder is able to use the data generated during profile runs using the `-profile` option to guide the compilation process,

producing optimal performance with a minimum increase in code size.

Here is a demonstration of the profiler using the HelloWorld example presented in § 2.4. First, it is built using the `-profile` option:

```
> jamaicabuilder -cp classes -profile -interpret HelloWorld
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/HelloWorld__.c
 + tmp/HelloWorld__.h
 * C compiling 'tmp/HelloWorld__.c'
 + tmp/HelloWorld__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                        initial               max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                256MB
GC data:                128KB                 16MB
TOTAL:                 3328KB                343MB
```

The generated executable HelloWorld now prints the profiling information after execution. The output may look like this:[1]

```
> ./HelloWorld 10000
              Hello        World!
           Hello       World!
        Hello     World!
     Hello    World!
   Hello   World!
 Hello  World!
 [...]
              Hello        World!
           Hello       World!
        Hello     World!
     Hello    World!
   Hello   World!
 Hello  World!
 [...]
Start writing profile data into file 'HelloWorld.prof'
 Write threads data...
 Write invocation data...
```

[1] For better results, we run the application with the command line argument `10000` such that startup code does not dominate.

```
Done writing profile data
```

5.1.2 Using the profiling VM

Alternatively, in simple cases, the profile can also be created using the `jamaicavmp` command on the host without first building a stand-alone executable:

```
> jamaicavmp HelloWorld 10000
              Hello        World!
           Hello       World!
        Hello     World!
     Hello    World!
   Hello   World!
 Hello  World!
 [...]
              Hello        World!
           Hello       World!
        Hello     World!
     Hello    World!
   Hello   World!
 Hello  World!
 [...]
Start writing profile data into file 'HelloWorld.prof'
 Write threads data...
 Write invocation data...
Done writing profile data
```

The use of `jamaicavmp` is subject to the following restrictions:

- It can generate a profile for the host only.
- Setting builder options for the application to be profiled is not possible.

5.1.3 Dumping a profile via network

If the application does not exit or writing a profile is very slow on the target, you can request a profile dump with the `jamaica_remoteprofile` command. You need to set the `jamaica.profile_request_port` property when building the application or using the profiling VM to an open TCP/IP port and then request a dump remotely:

```
> jamaica_remoteprofile target port
DUMPING...
DONE.
```

In the above command, *target* denotes the IP address or host name of the target system. By default, the profile is written on the target to a file with the name of the main class and the suffix `.prof`. You can change the file name with the `-file` option or you can send the profile over the network and write it to the file system (with an absolute path or relative to the current directory) of the host with the `-net` option:

```
> jamaica_remoteprofile -net=filename target port
```

5.1.4 Creating a micro profile

To speed up the performance of critical sections in the application, you can use micro profiles that only contain profiling information of such a section (see § 5.2.2). You need to reset the profile just before the critical part is executed and dump a profile directly after. To reset a profile, you can use the command `jamaica_remoteprofile` with the `-reset` option:

```
> jamaica_remoteprofile -reset target port
```

5.2 Using a profile with the Builder

Having collected the profiling data, the Jamaica Compiler can create a compiled version of the application using the profile information. This compiled version benefits from profiling information in several ways:

- Compilation is limited to the most time critical methods, keeping non-critical methods in smaller interpreted byte-code format.
- Method inlining prefers inlining of calls that have shown to be executed most frequently during the profiling run.
- Profiling information also collects information on the use of reflection, so an application that cannot use smart linking due to reflection can profit from smart linking even without manually listing all classes referenced via reflection.

5.2.1 Building with a profile

The builder option `-useProfile` is used to select the generated profiling data:

```
> jamaicabuilder -cp classes -useProfile HelloWorld.prof HelloWorld
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__V6b08cb32f992bc23__.c
 [...]
 + tmp/HelloWorld__.c
 + tmp/HelloWorld__.h
 * C compiling 'tmp/HelloWorld__.c'
 [...]
 + tmp/HelloWorld__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                       initial                max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                256MB
GC data:                128KB                 16MB
TOTAL:                 3328KB                343MB
```

Due to the profile-guided optimizations performed by the compiler, the runtime performance of the application built using a profile as shown usually exceeds the performance of a fully compiled application. Furthermore, the memory footprint is significantly smaller and the modify-compile-run cycle time is usually significantly shorter as well since only a small fraction of the application needs to be compiled. It is not necessary to re-generate profile data after every modification.

5.2.2 Building with multiple profiles

You can use several profiles to improve the performance of your application. There are two possibilities to specify profiles that behave in a different way.

First you can just concatenate two profile files or dump a profile several times into the same file which will just behave as if the profiles were recorded sequentially. You can add a profile for a new feature this way.

If you want to favor a profile instead, e.g. a micro profile for startup or a performance critical section as described in § 5.1.4, you can specify the profile with another `-useProfile` option. In this case, all profiles are normalized before they are concatenated, so highly rated methods in a short-run micro profile are more likely to be compiled.

5.3 Interpreting the profiling output

When running in profiling mode, the VM collects data to create an optimized application but can also be interpreted manually to find memory leaks or time consuming methods. You can make Jamaica collect information about performance, memory requirements etc.

To collect additional information, you have to set the property `jamaica.profile_groups` to select one or more profiling groups. The default value is `builder` to collect data used by the builder. You can set the property to the values `builder`, `memory`, `speed`, `all` or a comma separated combination of those. Example:

```
> jamaicavmp -cp classes \
>   -Djamaica.profile_groups=builder,speed \
>   HelloWorld 10000
```

! The format of the profile file is likely to change in future versions of Jamaica Builder.

5.3.1 Format of the profile file

Every line in the profiling output starts with a keyword followed by space separated values. The meaning of these values depends on the keyword.

For a better overview, the corresponding values in different lines are aligned as far as possible and words and signs that improve human reading are added. Here for every keyword the additional words and signs are omitted and the values are listed in the same order as they appear in the text file.

Keyword: `BEGIN_PROFILE_DUMP` **Groups:** all

Values

1. unique dump ID

Keyword: `END_PROFILE_DUMP` **Groups:** all

Values

1. unique dump ID

Keyword: `HEAP_REFS` **Groups:** `memory`

Values

1. total number of references in object attributes
2. total number of words in object attributes
3. relative number of references in object attributes

Keyword: `HEAP_USE` **Groups:** `memory`

Values

1. total number of currently allocated objects of this class
2. number of blocks needed for one object of this class
3. block size in bytes
4. number of bytes needed for all objects of this class
5. relative heap usage of objects of this class
6. total number of objects of this class organized in a tree structure
7. relative number of objects of this class organized in a tree structure
8. name of the class

Keyword: `INSTANTIATION_COUNT` **Groups:** `memory`

Values

1. total number of instantiated objects of this class
2. number of blocks needed for one object of this class
3. number of blocks needed for all objects of this class
4. number of bytes needed for all objects of this class
5. total number of objects of this class organized in a tree structure
6. relative number of objects of this class organized in a tree structure
7. class loader that loaded the class
8. name of the class

Keyword: `PROFILE` **Groups:** `builder`

Values

1. total number of bytecodes executed in this method
2. relative number of bytecodes executed in this method
3. signature of the method
4. class loader that loaded the class of the method

Keyword: `PROFILE_CLASS_USED_VIA_REFLECTION` **Groups:** `builder`

Values

1. name of the class used via reflection

Keyword: `PROFILE_CYCLES` **Groups:** `speed`

Values

1. total number of processor cycles spent in this method (if available on the target)
2. signature of the method

Keyword: `PROFILE_INVOKE` **Groups:** `builder`

Values

1. number of calls from caller method to called method
2. bytecode position of the call within the method
3. signature of the caller method
4. signature of the called method

Keyword: `PROFILE_INVOKE_CYCLES` **Groups:** `speed`

Values

1. number of processor cycles spent in the called method

2. bytecode position of the call within the method
3. signature of the caller method
4. signature of the called method

Keyword: `PROFILE_NATIVE` **Groups:** all

Values

1. total number of calls to the native method
2. relative number of calls to the native method
3. signature of the called native method

Keyword: `PROFILE_NEWARRAY` **Groups:** `memory`

Values

1. number of calls to array creation within a method
2. bytecode position of the call within the method
3. signature of the method

Keyword: `PROFILE_THREAD` **Groups:** `memory`, `speed`

Values

1. current Java priority of the thread
2. total amount of CPU cycles in this thread
3. relative time in interpreted code
4. relative time in compiled code
5. relative time in JNI code
6. relative time in garbage collector code
7. required C stack size
8. required Java stack size

Keyword: `PROFILE_THREADS` **Groups:** `builder`

Values

1. maximum number of concurrently used threads

Keyword: `PROFILE_THREADS_JNI` **Groups:** `builder`

Values

1. maximum number of threads attached via JNI

Keyword: `PROFILE_VERSION` **Groups:** all

Values

1. version of Jamaica the profile was created with

5.3.2 Example

We can sort the profiling output to find the application methods where most of the execution time is spent. Under Unix, the 25 methods which use the most execution time (in number of bytecode instructions) can be found with the following command:

```
> grep PROFILE: HelloWorld.prof | sort -rn -k2 | head -n25
PROFILE: 7198992 (27%)  sun/nio/cs/UTF_8$Encoder.encodeArrayLo...
PROFILE: 1722630 (6%)   java/lang/String.getChars(II[CI)V [boot]
PROFILE: 1072566 (4%)   java/lang/AbstractStringBuilder.append...
PROFILE: 1060053 (3%)   java/io/BufferedWriter.write(Ljava/lan...
PROFILE: 781118 (2%)    java/nio/Buffer.position(I)Ljava/nio/B...
PROFILE: 720036 (2%)    sun/nio/cs/StreamEncoder.writeBytes()V...
PROFILE: 700035 (2%)    sun/nio/cs/StreamEncoder.write([CII)V ...
PROFILE: 633824 (2%)    java/lang/String.length()I [boot]
PROFILE: 614990 (2%)    java/lang/String.substring(II)Ljava/la...
PROFILE: 560457 (2%)    java/nio/charset/CharsetEncoder.encode...
PROFILE: 520026 (1%)    sun/nio/cs/StreamEncoder.implWrite([CI...
PROFILE: 481056 (1%)    java/nio/Buffer.<init>(IIII)V [boot]
PROFILE: 480384 (1%)    java/nio/ByteBuffer.arrayOffset()I [boot]
PROFILE: 451926 (1%)    java/lang/AbstractStringBuilder.value(...
PROFILE: 450019 (1%)    HelloWorld.main([Ljava/lang/String;)V ...
PROFILE: 400912 (1%)    java/lang/AbstractStringBuilder.capaci...
PROFILE: 400020 (1%)    java/io/BufferedWriter.flushBuffer()V ...
PROFILE: 360378 (1%)    java/nio/CharBuffer.arrayOffset()I [boot]
PROFILE: 360018 (1%)    java/io/PrintStream.write([BII)V [boot]
PROFILE: 320704 (1%)    java/nio/Buffer.limit(I)Ljava/nio/Buff...
```

```
PROFILE: 300315 (1%)    sun/nio/cs/UTF_8.updatePositions(Ljava...
PROFILE: 280014 (1%)    sun/nio/cs/StreamEncoder.flushBuffer()...
PROFILE: 266867 (1%)    java/lang/AbstractStringBuilder.expand...
PROFILE: 260273 (0%)    java/nio/CharBuffer.<init>(IIII[CI)V [...
PROFILE: 240414 (0%)    java/lang/StringBuilder.append(Ljava/l...
```

In this small example program, it is not a surprise that nearly all execution time is spent in methods that are required for writing the output to the screen. The dominant function is `UTF_8$Encoder.encodeArrayLoop` from the OpenJDK classes included in Jamaica, which is used while converting Java's unicode characters to the platform's UTF-8 encoding. Also important is the time spent in `AbstractStringBuilder.append`. Calls to the AbstractStringBuilder methods have been generated automatically by the `jamaicac` compiler for string concatenation expressions using the '+'-operator.

On systems that support a CPU cycle counter, the profiling data also contains a cumulative count of the number of processor cycles spent in each method. This information is useful to obtain a more high-level view on where the runtime performance was spent.

The CPU cycle profiling information is contained in lines starting with the tag `PROFILE_CYCLES:`. A similar command line can be used to find the methods that cumulatively require most of the execution time:

```
> grep PROFILE_CYCLES: HelloWorld.prof | sort -rn -k2 | head -n25
PROFILE_CYCLES: 2735009892      java/io/PrintStream.println(Lj...
PROFILE_CYCLES: 1599738722      java/io/BufferedWriter.flushBu...
PROFILE_CYCLES: 1533465661      java/io/PrintStream.print(Ljav...
PROFILE_CYCLES: 1527745949      java/io/OutputStreamWriter.wri...
PROFILE_CYCLES: 1524918219      java/io/PrintStream.write(Ljav...
PROFILE_CYCLES: 1505911987      sun/nio/cs/StreamEncoder.write...
PROFILE_CYCLES: 1426767007      sun/nio/cs/StreamEncoder.implW...
PROFILE_CYCLES: 1179239368      java/io/PrintStream.newLine()V...
PROFILE_CYCLES: 1052899177      java/nio/charset/CharsetEncode...
PROFILE_CYCLES: 953473477       sun/nio/cs/UTF_8$Encoder.encod...
PROFILE_CYCLES: 871471771       sun/nio/cs/UTF_8$Encoder.encod...
PROFILE_CYCLES: 753616560       java/io/OutputStreamWriter.flu...
PROFILE_CYCLES: 735288717       sun/nio/cs/StreamEncoder.flush...
PROFILE_CYCLES: 681592360       sun/nio/cs/StreamEncoder.implF...
PROFILE_CYCLES: 642236787       sun/nio/cs/StreamEncoder.write...
PROFILE_CYCLES: 623178894       java/lang/StringBuilder.append...
PROFILE_CYCLES: 586412561       java/lang/AbstractStringBuilde...
PROFILE_CYCLES: 528166611       java/lang/Class.desiredAsserti...
PROFILE_CYCLES: 367374042       java/io/PrintStream.write([BII...
PROFILE_CYCLES: 302631112       java/io/FileOutputStream.write...
PROFILE_CYCLES: 274009758       java/io/FileOutputStream.write...
PROFILE_CYCLES: 253890062       java/nio/CharBuffer.wrap([CII)...
PROFILE_CYCLES: 252562489       java/io/Writer.write(Ljava/lan...
```

```
PROFILE_CYCLES: 229642029       java/lang/ClassLoader$ExtClass...
PROFILE_CYCLES: 211995274       java/lang/String.getChars(II[C...
```

The report is cumulative. It shows more clearly how much time is spent in which method. The method `println(String)` of class `java.io.PrintStream` dominates the program. The main method of a program is not included in the `PROFILE_CYCLES`.

The cumulative cycle counts can now be used as a basis for a top-down optimization of the application execution time.

Chapter 6

Reducing Footprint and Memory Usage

This chapter is a hands-on tutorial that shows how to reduce an application's footprint and RAM demand, while also achieving optimal runtime performance. As example application we use Pendragon Software's embedded CaffeineMark (tm) 3.0. The class files for this benchmark are part of the JamaicaVM Tools installation. See § 2.4.

6.1 Compilation

JamaicaVM Builder compiles bytecode to machine code, which is typically about 20 to 30 times faster than interpreted code. (This is called *static compilation*.) However, due to the fact that Java bytecode is very compact compared to machine code on CISC or RISC machines, compiled code is significantly larger than bytecode.

Therefore, in order to improve the performance of an application, only those bytecodes that contribute most to the overall runtime should be compiled to machine code in order to achieve satisfactory runtime. This is done using a profile and was discussed in the previous chapter (§ 5). While using a profile usually offers the best compromise between footprint and performance, JamaicaVM Builder also provides other modes of compilation. They are discussed in the following sections.

6.1.1 Suppressing Compilation

The builder option `-interpret` turns compilation of bytecode off. The created executable will be a standalone program containing both bytecode of the applica-

tion and the virtual machine executing the bytecode.

```
> jamaicabuilder -cp classes CaffeineMarkEmbeddedApp -interpret \
>   -destination=caffeine_interpret
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/caffeine_interpret__.c
 + tmp/caffeine_interpret__.h
 * C compiling 'tmp/caffeine_interpret__.c'
 + tmp/caffeine_interpret__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                      initial               max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                256MB
GC data:                128KB                 16MB
TOTAL:                 3328KB                343MB
```

The size of the created binary may be inspected, for example, with a shell command to list directories. We use `ls -sk` *file*, which displays the file size in 1024 Byte units. It is available on Unix systems. On Windows, `dir` may be used instead.

```
> ls -sk caffeine_interpret
10232   caffeine_interpret
```

The runtime performance for the built application is slightly better compared to using `jamaicavm_slim`.

```
> ./caffeine_interpret
Sieve score = 1514 (98)
Loop score = 1265 (2017)
Logic score = 1572 (0)
String score = 2127 (708)
Float score = 1280 (185)
Method score = 1150 (166650)
Overall score = 1453

> jamaicavm_slim CaffeineMarkEmbeddedApp
Sieve score = 1326 (98)
Loop score = 960 (2017)
Logic score = 1545 (0)
```

```
String score = 1189 (708)
Float score = 1107 (185)
Method score = 1063 (166650)
Overall score = 1183
```

Better performance will be achieved by compilation as shown in the sections below.

6.1.2 Using Default Compilation

If none of the options `interpret`, `compile`, or `useProfile` is specified, default compilation is used. The default means that a pre-generated profile will be used for the system classes, and all application classes will be compiled fully. This usually results in good performance for small applications, but it causes substantial code size increase for larger applications and it results in slow execution of applications that use the system classes in a way different than recorded in the system profile.

```
> jamaicabuilder -cp classes CaffeineMarkEmbeddedApp \
>   -destination=caffeine
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__Ve96705067e690195__.c
 [...]
 + tmp/caffeine__.c
 + tmp/caffeine__.h
 * C compiling 'tmp/caffeine__.c'
 [...]
 + tmp/caffeine__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                        initial               max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                 256MB
GC data:                128KB                  16MB
TOTAL:                 3328KB                 343MB

> ls -sk caffeine
11012   caffeine
```

The performance of this example is dramatically better than the performance of the interpreted version.

```
> ./caffeine
Sieve score = 31591 (98)
Loop score = 155607 (2017)
Logic score = 895517 (0)
String score = 2278 (708)
Float score = 33454 (185)
Method score = 23018 (166650)
Overall score = 44458
```

6.1.3 Using a Custom Profile

Generation of a profile for compilation is a powerful tool for creating small applications with fast turn-around times. The profile collects information on the runtime behavior of an application, guiding the compiler in its optimization process and in the selection of which methods to compile and which methods to leave in compact bytecode format.

To generate the profile, we first have to create a profiling version of the applications using the builder option `profile` (see § 5) or using the command `jamaicavmp`:

```
> jamaicavmp CaffeineMarkEmbeddedApp
Sieve score = 712 (98)
Loop score = 608 (2017)
Logic score = 691 (0)
String score = 1021 (708)
Float score = 611 (185)
Method score = 524 (166650)
Overall score = 678
Start writing profile data into file 'CaffeineMarkEmbeddedApp.prof'
 Write threads data...
 Write invocation data...
Done writing profile data
```

This profiling run also illustrates the runtime overhead of the profiling data collection: the profiling run is significantly slower than the interpreted version.

Now, an application can be compiled using the profiling data that was stored in file `CaffeineMarkEmbeddedApp.prof`:

```
> jamaicabuilder -cp classes \
>  -useProfile=CaffeineMarkEmbeddedApp.pro \
>  CaffeineMarkEmbeddedApp -destination=caffeine_useProfile10
Reading configuration from
```

```
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__V6a23ba6367e1bd7a__.c
 [...]
 + tmp/caffeine_useProfile10__.c
 + tmp/caffeine_useProfile10__.h
 * C compiling 'tmp/caffeine_useProfile10__.c'
 [...]
 + tmp/caffeine_useProfile10__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                        initial               max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                256MB
GC data:                128KB                 16MB
TOTAL:                 3328KB                343MB
```

The resulting application is only slightly larger than the interpreted version, but, as expected, the runtime score is better than when using the default profile:

```
> ls -sk caffeine_useProfile10
10480   caffeine_useProfile10

> ./caffeine_useProfile10
Sieve score = 31531 (98)
Loop score = 160692 (2017)
Logic score = 895567 (0)
String score = 3986 (708)
Float score = 33079 (185)
Method score = 23020 (166650)
Overall score = 48959
```

When a profile is used to guide the compiler, by default 10% of the methods executed during the profile run are compiled. This results in a moderate code size increase compared with fully interpreted code and results in a run-time performance very close to or typically even better than fully compiled code. Using the builder option `percentageCompiled`, this default setting can be adjusted to any value from 0% to 100%. Best results are usually achieved with a value from 10% to 30%, where a higher value leads to a larger footprint. Note that setting the value to 100% is not the same as setting the option `compile` (see § 6.1.5), since using a profile only compiles those methods that are executed during the profiling

run. Methods not executed during the profiling run will not be compiled when `useProfile` is used.

Entries in the profile can be edited manually, for example to enforce compilation of a method that is performance critical. For example, the profile generated for this example contains the following entry for the method `size()` of class `java.util.Vector`.

```
PROFILE: 64 (0%)        java/util/Vector.size()I
```

To enforce compilation of this method even when `percentageCompiled` is not set to 100%, the profiling data can be changed to a higher value, e.g.,

```
PROFILE: 1000000 (0%)        java/util/Vector.size()I
```

6.1.4 Code Optimization by the C Compiler

Enabling C compiler optimizations for code size or execution speed can have an important effect on the the size and speed of the application. These optimizations are enabled via setting the command line options `-optimize=size` or `-optimize=speed`, respectively. Note that `speed` is normally the default.[1] For comparison, we build the caffeine example optimizing for `size`.

```
> jamaicabuilder -cp classes \
>   -useProfile=CaffeineMarkEmbeddedApp.prof \
>   -optimize=size CaffeineMarkEmbeddedApp \
>   -destination=caffeine_useProfile10_size
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'size'
 + tmp/PKG__Ved90dd4822aec427__.c
 [...]
 + tmp/caffeine_useProfile10_size__.c
 + tmp/caffeine_useProfile10_size__.h
 * C compiling 'tmp/caffeine_useProfile10_size__.c'
 [...]
 + tmp/caffeine_useProfile10_size__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                       initial                max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
```

[1] To check the default, invoke `jamaicabuilder -help` or inspect the builder status messages.

```
Heap Size:              2048KB                  256MB
GC data:                 128KB                   16MB
TOTAL:                  3328KB                  343MB
```

Code size and performance depend strongly on the C compiler that is employed and may even show anomalies such as better runtime performance for the version optimized for smaller code size. We get these results:

```
> ls -sk caffeine_useProfile10_size
10280   caffeine_useProfile10_size

> ./caffeine_useProfile10_size
Sieve score = 27124 (98)
Loop score = 105519 (2017)
Logic score = 717829 (0)
String score = 3926 (708)
Float score = 33454 (185)
Method score = 23063 (166650)
Overall score = 42888
```

6.1.5 Full Compilation

Full compilation can be used when no profiling information is available and code size and build time are not important issues.

! Fully compiling an application leads to very poor turn-around times and may require significant amounts of memory during the C compilation phase. We recommend compilation be used only through profiling as described above.

To compile the complete application, the option `compile` is set:

```
> jamaicabuilder -cp classes -compile CaffeineMarkEmbeddedApp \
>   -destination=caffeine_compiled
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__Ve0cfdbb45ac4ab9a__.c
 [...]
 + tmp/caffeine_compiled__.c
 + tmp/caffeine_compiled__.h
 * C compiling 'tmp/caffeine_compiled__.c'
 [...]
 + tmp/caffeine_compiled__nc.o
 * linking
```

```
 * stripping
Application memory demand will be as follows:
                        initial                max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                 256MB
GC data:                128KB                  16MB
TOTAL:                 3328KB                 343MB
```

The resulting binary is very large. The performance of the compiled version is significantly better than the interpreted version. However, even though all code was compiled, the performance of the versions created using profiles is not matched. This is due to poor cache behavior caused by the large footprint.

```
> ls -sk caffeine_compiled
44896   caffeine_compiled

> ./caffeine_compiled
Sieve score = 31499 (98)
Loop score = 155613 (2017)
Logic score = 896456 (0)
String score = 1203 (708)
Float score = 33864 (185)
Method score = 23006 (166650)
Overall score = 40036
```

Full compilation is only feasible in combination with the code size optimizations discussed in the sequel. Experience shows that using a custom profile is superior in almost all situations.

6.2 Smart Linking

The JamaicaVM Builder can remove unused bytecode from an application. This is called *smart linking* and reduces the footprint of both interpreted and statically compiled code. By default, only a modest degree of smart linking is used: unused classes and methods of classes are removed, unless that code is explicitly included with either of the options `-includeClasses` or `-includeJAR`. For more information, see the builder option `-smart`.

Additional optimizations are possible if the builder knows for sure that the application that is compiled is closed, i.e., all classes of the application are built-in and the application does not use dynamic class loading to add any additional code. These additional optimizations include static binding and inlining for virtual

method calls if the called method is not redefined by any built-in class. The builder can be instructed to perform these optimizations by setting the option `-closed`.

In the Caffeine benchmark application, dynamic class loading is not used, so we can enable closed application optimizations by setting `-closed`:

```
> jamaica -cp classes -closed \
>  -useProfile=CaffeineMarkEmbeddedApp.prof \
>  CaffeineMarkEmbeddedApp \
>  -destination=caffeine_useProfile10_closed
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__Vc61a8793805c8445__.c
 [...]
 + tmp/caffeine_useProfile10_closed__.c
 + tmp/caffeine_useProfile10_closed__.h
 * C compiling 'tmp/caffeine_useProfile10_closed__.c'
 [...]
 + tmp/caffeine_useProfile10_closed__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                      initial               max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                256MB
GC data:                128KB                 16MB
TOTAL:                 3328KB                343MB

> ls -sk caffeine_useProfile10_closed
10324   caffeine_useProfile10_closed
```

The effect on the code size is favourable. Also, the resulting runtime performance is significantly better for code that requires frequent virtual method calls. Consequently, the results of the Method test in the Caffeine benchmark are improved when closed application optimizations are enabled:

```
> ./caffeine_useProfile10_closed
Sieve score = 31563 (98)
Loop score = 161495 (2017)
Logic score = 895685 (0)
String score = 3992 (708)
Float score = 33216 (185)
Method score = 69266 (166650)
Overall score = 58942
```

6.3 API Library Classes and Resources

The footprint of an application can be further reduced by excluding resources such as language locales and time zone information, which contain a fair amount of data, and their associated library classes.

For our example application, there is no need for supporting network protocols or language locales. Furthermore, neither graphics nor fonts are needed. Consequently, we can set all of `protocols`, `locales`, `graphics` and `fonts` to the empty set. Time zone support is not required either, and we include only a single time zone. The resulting call to build the application is as follows:

```
> jamaicabuilder -cp classes -closed \
>  -useProfile=CaffeineMarkEmbeddedApp.prof \
>   -setProtocols=none -setLocales=none \
>   -setGraphics=none -setFonts=none \
>   -setTimeZones=Europe/Berlin \
>   CaffeineMarkEmbeddedApp -destination=caffeine_nolibs
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__Vb7d1f2cadcbdca23__.c
 [...]
 + tmp/caffeine_nolibs__.c
 + tmp/caffeine_nolibs__.h
 * C compiling 'tmp/caffeine_nolibs__.c'
 [...]
 + tmp/caffeine_nolibs__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                      initial               max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                256MB
GC data:                128KB                 16MB
TOTAL:                 3328KB                343MB

> ls -sk caffeine_nolibs
2892    caffeine_nolibs
```

A huge part of the class library code could be removed by the Jamaica Builder so that the resulting application is significantly smaller than in the previous examples.

6.4 RAM Usage

In many embedded applications, the amount of random access memory (RAM) required is even more important than the application performance and its code size. Therefore, a number of means to control the application RAM demand are available in Jamaica. RAM is required for three main purposes:

1. Memory for application data structures, such as objects or arrays allocated at runtime.

2. Memory required to store internal data of the VM, such as representations of classes, methods, method tables, etc.

3. Memory required for each thread, such as Java and C stacks.

Needless to say that Item 1 is predominant for an application's use of RAM space. This includes choosing appropriate classes from the standard library. For memory critical applications, the used data structures should be chosen with care. The memory overhead of a single object allocated on the Jamaica heap is relatively small: typically three machine words are required for internal data such as the garbage collection state, the object's type information, a monitor for synchronization and memory area information. See § 9 for details on memory areas.

Item 2 means that an application that uses fewer classes will also have a lower memory demand. Consequently, the optimizations discussed in the previous sections (§ 6.2 and § 6.3) have a knock-on effect on RAM demand! Memory needed for threads (Item 3) can be controlled by configuring the number of threads available to the application and the stack sizes.

6.4.1 Measuring RAM Demand

The amount of RAM actually needed by an application can be determined by setting the builder option `analyze`. Apart from setting this option, it is important that exactly the same arguments are used as in the final version. Here `analyze` is set to '`1`', which yields an accuracy of 1%:

```
> jamaicabuilder -cp classes -analyze=1 -closed \
>  -useProfile=CaffeineMarkEmbeddedApp.prof \
>   CaffeineMarkEmbeddedApp -destination=caffeine_analyze
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__Vc61a8793805c8445__.c
```

```
 [...]
 + tmp/caffeine_analyze__.c
 + tmp/caffeine_analyze__.h
 * C compiling 'tmp/caffeine_analyze__.c'
 [...]
 + tmp/caffeine_analyze__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                       initial               max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                256MB
GC data:                128KB                 16MB
TOTAL:                 3328KB                343MB
```

Running the resulting application will print the amount of RAM memory that was required during the execution:

```
> ./caffeine_analyze
Sieve score = 15951 (98)
Loop score = 15627 (2017)
Logic score = 897203 (0)
String score = 39 (708)
Float score = 8131 (185)
Method score = 69469 (166650)
Overall score = 13044

### Recommended heap size: 3355K (contiguous memory).
### Application used at most 2029984 bytes for reachable objects
on the Java heap
### (accuracy 1%).
###
### Reserved memory is set to 10%.  To obtain lower memory bounds
### or worst-case GC overhead, set reserved memory to 0.
###
### Worst case allocation overhead:
###     heapSize        dynamic GC      const GC work
###     10489K          6               3
###     8059K           7               4
###     6698K           8               4
###     5848K           9               4
###     5259K           10              4
###     4506K           12              5
###     4054K           14              5
###     3741K           16              6
###     3522K           18              6
###     3355K           20              7
```

```
###     3122K           24              8
###     2968K           28              9
###     2861K           32              10
###     2777K           36              11
###     2712K           40              12
###     2619K           48              14
###     2555K           56              17
###     2510K           64              19
###     2403K           96              27
###     2352K           128             36
###     2300K           192             53
###     2276K           256             69
###     2253K           384             100
```

The memory analysis report begins with a recommended heap size and the actual memory demand. The latter is the maximum of simultaneously reachable objects during the entire program run. The JamaicaVM garbage collector needs more memory than the actual memory demand to do its work. The overhead depends on the GC mode and the amount of collection work done per allocation. In dynamic mode, which is the default, 20 units of collection work per allocation are recommended, which leads to a memory overhead. Overheads for various garbage collection work settings are shown in the table printed by the analyze mode. For more information on heap size analysis and the Builder option `-analyze`, see § 7.2.

6.4.2 Memory Required for Threads

To reduce memory other than the Java heap, one must reduce the stack sizes and the number of threads that will be created for the application. This can be done in the following ways.

6.4.2.1 Reducing Stack Sizes

The Java stack size can be reduced via option `javaStackSize` to a lower value than the default (typically 20K). To reduce the size to 4K, `javaStackSize=4K` can be used. The C stack size can be set accordingly with `nativeStackSize`.

6.4.2.2 Disabling the Finalizer Thread

A Java application typically uses one thread that is dedicated to running the finalization methods (`finalize()`) of objects that were found to be unreachable by the garbage collector. An application that does not allocate any such objects may not need the finalizer thread. The priority of the finalizer thread can

be adjusted through the option `finalizerPri`. Setting the priority to zero (`-finalizerPri=0`) deactivates the finalizer thread completely.

Note that deactivating the finalizer thread may cause a memory leak since any objects that have a `finalize()` method can no longer be reclaimed. Similarly, weak, soft and phantom references rely on the presence of a finalizer. If the resources available on the target system do not permit the use of a finalizer thread, the application may execute the `finalize()` method explicitly by frequent calls to `Runtime.runFinalization()`. This will also permit the use of weak, soft and phantom references if no finalizer thread is present.

6.4.2.3 Setting the Number of Threads

The number of threads available for the application can be set using the option `numThreads`. The default setting for this option is two, which is enough for the finalizer thread and the main thread of the application.

If the finalizer thread is deactivated and no new threads are started by the application, the number of threads can be reduced to one by using the setting `-numThreads=1`.

If profiling information was collected and is provided via the `useProfile` option, the number of threads provided to the `numThreads` option is checked to ensure it is at least the number of threads that was required during the profiling run. If not, a warning with the minimum number of threads during the profiling run will be displayed. This information can be used to adjust the number of threads to the minimum required by the application.

6.4.2.4 Disabling Time Slicing

On non-realtime systems that do not strictly respect thread priorities, Jamaica uses one additional thread to allow time slicing between threads. On realtime systems, this thread can be used to enforce round-robin scheduling of threads of equal priorities.

On systems with tight memory demand, the thread required for time-slicing can be deactivated by setting the size of the time slice to zero using the option `-timeSlice=0ns`. In an application that uses threads of equal priorities, explicit calls to the method `Thread.yield()` are required to permit thread switches to another thread of the same priority if the time slicing thread is disabled.

The number of threads set by the option `-numThreads` does not include the time slicing thread. Unlike when disabling the finalizer thread, which is a Java thread, when the time slicing thread is disabled, the argument to `-numThreads` should not be changed.

6.4.2.5 Disabling the Memory Reservation Thread

The memory reservation thread is a low priority thread that continuously tries to reserve memory up to a specified threshold. This reserved memory is used by all other threads. As long as reserved memory is available no GC work needs to be done. This is especially efficient for applications that have long pause times with little or no activity that are preempted by sudden activities that require a burst of memory allocation.

On systems with tight memory demand, the thread required for memory reservation can be deactivated by setting `-reservedMemory=0`.

6.4.2.6 Disabling Signal Handlers

The default handlers for the POSIX signals can be turned off by setting properties with the option `XdefineProperty`. The POSIX signals are SIGINT, SIGQUIT and SIGTERM. The properties are described in § 13.4. To turn off the signal handlers, the properties `jamaica.no_sig_int_handler`, `jamaica.no_sig_quit_handler` and `jamaica.no_sig_term_handler` should be set to `true`.

6.4.2.7 The Example Continued

Applying this to our example application, we can reduce the Java stack to 4K, deactivate the finalizer thread, set the number of threads to 1, disable the time slicing thread and the memory reservation thread and turn off the signal handlers:

```
> jamaicabuilder -cp classes -closed \
>   -useProfile=CaffeineMarkEmbeddedApp.prof \
>   -setLocales=none -setProtocols=none \
>   -setGraphics=none -setFonts=none \
>   -setTimeZones=Europe/Berlin \
>   -javaStackSize=4K -finalizerPri=0 -numThreads=1 \
>   -timeSlice=0ns -reservedMemory=0 \
>   -XdefineProperty="jamaica.no_sig_int_handler=true \
>   jamaica.no_sig_quit_handler=true \
>   jamaica.no_sig_term_handler=true" \
>   CaffeineMarkEmbeddedApp -destination=caffeine_nolibs_js_fP_tS
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__V8a9d89953725302b__.c
 [...]
 + tmp/caffeine_nolibs_js_fP_tS__.c
```

```
 + tmp/caffeine_nolibs_js_fP_tS__.h
 * C compiling 'tmp/caffeine_nolibs_js_fP_tS__.c'
 [...]
 + tmp/caffeine_nolibs_js_fP_tS__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                       initial               max
Thread C    stacks:     128KB (=   1* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:    4096B  (=   1*4096B ) 2044KB (= 511*4096B )
Heap Size:             2048KB                 256MB
GC data:                128KB                  16MB
TOTAL:                 2308KB                 337MB

> ls -sk caffeine_nolibs_js_fP_tS
2892    caffeine_nolibs_js_fP_tS
```

The additional options have little effect on the application size itself compared to the earlier version. However, the RAM allocated by the application was reduced significantly.

6.4.3 Memory Required for Line Numbers

An important advantage of programming in the Java language are the accurate error messages. Runtime exceptions contain a complete stack trace with line number information on where the problem occurred. This information, however, needs to be stored in the application and be available at runtime.

After the debugging of an application is finished, the memory demand of an application may be further reduced by removing this information. The builder option `XignoreLineNumbers` can be set to suppress it. Continuing the example from the previous section, we can further reduce the RAM demand by setting this option:

```
> jamaicabuilder -cp classes -closed \
>   -useProfile=CaffeineMarkEmbeddedApp.prof \
>   -setLocales=none -setProtocols=none -setGraphics=none \
>   -setFonts=none -setTimeZones=Europe/Berlin \
>   -javaStackSize=4K -finalizerPri=0 \
>   -numThreads=1 -timeSlice=0ns -reservedMemory=0 \
>   -XdefineProperty="jamaica.no_sig_int_handler=true \
>   jamaica.no_sig_quit_handler=true \
>   jamaica.no_sig_term_handler=true" \
>   CaffeineMarkEmbeddedApp -XignoreLineNumbers \
>   -destination=caffeine_nolibs_js_fP_tS_nL
Reading configuration from
```

```
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__V8a9d89953725302b__.c
 [...]
 + tmp/caffeine_nolibs_js_fP_tS_nL__.c
 + tmp/caffeine_nolibs_js_fP_tS_nL__.h
 * C compiling 'tmp/caffeine_nolibs_js_fP_tS_nL__.c'
 [...]
 + tmp/caffeine_nolibs_js_fP_tS_nL__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                        initial               max
Thread C    stacks:     128KB (=   1* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:    4096B  (=   1*4096B ) 2044KB (= 511*4096B )
Heap Size:             2048KB                 256MB
GC data:                128KB                  16MB
TOTAL:                 2308KB                 337MB
```

The size of the executable has shrunk since line number information is no longer present:

```
> ls -sk caffeine_nolibs_js_fP_tS_nL
2624    caffeine_nolibs_js_fP_tS_nL
```

By inspecting the builder output, we see that the initial memory demand reported by the builder was not reduced. The actual memory demand may be checked by repeating the build with the additional option `-analyze=1` and running the obtained executable:

```
> ./caffeine_analyze_nolibs_js_fP_tS_nL
Sieve score = 16089 (98)
Loop score = 15638 (2017)
Logic score = 897297 (0)
String score = 42 (708)
Float score = 8730 (185)
Method score = 69446 (166650)
Overall score = 13384

### Recommended heap size: 1100K (contiguous memory).
### Application used at most 777952 bytes for reachable objects
on the Java heap
### (accuracy 1%).
###
### Worst case allocation overhead:
```

```
###     heapSize        dynamic GC      const GC work
###     2629K           6               3
###     2196K           7               4
###     1919K           8               4
###     1731K           9               4
###     1593K           10              4
###     1407K           12              5
###     1290K           14              5
###     1206K           16              6
###     1146K           18              6
###     1100K           20              7
###     1034K           24              8
###     990K            28              9
###     958K            32              10
###     934K            36              11
###     915K            40              12
###     887K            48              14
###     868K            56              17
###     854K            64              19
###     822K            96              27
###     806K            128             36
###     790K            192             53
###     783K            256             69
###     776K            384             100
```

The actual memory demand was reduced to about one third compared to § 6.4.1. The score in analyze mode is significantly lower than the one of the production version. To conclude the example we verify that the score of the latter has not gone down as a result of the memory optimizations:

```
> ./caffeine_nolibs_js_fP_tS_nL
Sieve score = 32179 (98)
Loop score = 162196 (2017)
Logic score = 899582 (0)
String score = 3769 (708)
Float score = 32953 (185)
Method score = 69518 (166650)
Overall score = 58610
```

Chapter 7

Memory Management Configuration

JamaicaVM provides the only efficient hard-realtime garbage collector available for Java implementations on the marked today. This chapter will first explain how this garbage collection technology can be used to obtain the best results for applications that have soft-realtime requirements before explaining the more fine-grained tuning required for realtime applications.

7.1 Configuration for soft-realtime applications

For most non-realtime applications, the default memory management settings of JamaicaVM perform well: The heap size is set to a small starting size and is extended up to a maximum size automatically whenever the heap is not sufficient or the garbage collection work becomes too high. However, in some situations, some specific settings may help to improve the performance of a soft-realtime application.

7.1.1 Initial heap size

The default initial heap size is a small value. The heap size is increased on demand when the application exceeds the available memory or the garbage collection work required to collect memory in this small heap becomes too high. This means that an application that on startup requires significantly more memory than the initial heap size will see its startup time increased by repeated incremental heap size expansion.

The obvious solution here is to set the initial heap size to a value large enough for the application to start. The Jamaica builder option `heapSize` (see § 14) and

the virtual machine option `Xms`*size* can be employed to set a higher size.

Starting off with a larger initial heap not only prevents the overhead of incremental heap expansion, but it also reduces the garbage collection work during startup. This is because the garbage collector determines the amount of garbage collection work from the amount of free memory, and with a larger initial heap, the initial amount of free memory is larger.

7.1.2 Maximum heap size

The maximum heap size specified via builder option `maxHeapSize` (see § 14) and the virtual machine option `Xmx` should be set to the maximum amount of memory on the target system that should be available to the Java application. Setting this option has no direct impact on the performance of the application as long as the application's memory demand does not come close to this limit. If the maximum heap size is not sufficient, the application will receive an `OutOfMemoryError` at runtime.

However, it may make sense to set the initial heap size to the same value as the maximum heap size whenever the initial heap demand of the application is of no importance for the remaining system. Setting initial heap size and maximum heap size to the same value has two main consequences. First, as has been seen in § 7.1.1 above, setting the initial heap size to a higher value avoids the overhead of dynamically expanding the heap and reduces the amount of garbage collection work during startup. Second, JamaicaVM's memory management code contains some optimizations that are only applicable to a non-increasing heap memory space, so overall memory management overhead will be reduced if the same value is chosen for the initial and the maximum heap size.

7.1.3 Finalizer thread priority

Before the memory used by an object that has a `finalize` method can be reclaimed, this `finalize` method needs to be executed. A dedicated thread, the `FinalizerThread` executes these `finalize` methods and otherwise sleeps waiting for the garbage collector to find objects to be finalized.

In order to prevent the system from running out of memory, the `FinalizerThread` must receive sufficient CPU time. Its default priority is therefore set to 10, the highest priority a Java thread may have. Consequently, any thread with a lower priority will be preempted whenever an object is found to require finalization.

Selecting a lower finalizer thread priority may cause the finalizer thread to starve if a higher priority thread does not yield the CPU for a longer period of time. However, if it can be guaranteed that the finalizer thread will not starve,

system performance may be improved by running the finalizer thread at a lower priority. Then, a higher priority thread that performs memory allocation will not be preempted by finalizer thread execution.

This priority can be set to a lower value using the option `finalizerPri` of the builder or the environment variable `JAMAICAVM_FINALIZERPRI`. In an application that has sufficient idle CPU time in between urgent activities, a finalizer priority lower than the priority of all other threads may be sufficient.

7.1.4 Reserved memory

JamaicaVM's default behavior is to perform garbage collection work at memory allocation time. This ensures a fair accounting of the garbage collection work: Those threads with the highest allocation rate will perform correspondingly more garbage collection work.

However, this approach may slow down threads that run only occasionally and perform some allocation bursts, e.g., changing the input mask or opening a new window in a graphical user interface.

To avoid penalizing these time-critical tasks by allocation work, JamaicaVM uses a low priority memory reservation thread that runs to pre-allocate a given percentage of the heap memory. This reserved memory can then be allocated by any allocation bursts without the need to perform garbage collection work. Consequently, an application with bursts of allocation activity with sufficient idle time between these bursts will see an improved performance.

The maximum amount of memory that will be reserved by the memory reservation thread is given as a percentage of the total memory. The default value for this percentage is 10%. It can be set via the builder options `-reservedMemory` and `-reservedMemoryFromEnv`, or for the virtual machine via the environment variable `JAMAICAVM_RESERVEDMEMORY`.

An allocation burst that exceeds the amount of reserved memory will have to fall back to perform garbage collection work as soon as the amount of reserved memory is exceeded. This may occur if the maximum amount of reserved memory is less than the memory allocated during the burst or if there is too little idle time in between consecutive bursts such as when the reservation thread cannot catch up and reserve the maximum amount of memory.

For an application that cannot guarantee sufficient idle time for the memory reservation thread, the amount of reserved memory should not be set to a high percentage. Higher values will increase the worst case garbage collection work that will have to be performed on an allocation, since after the reserved memory was allocated, there is less memory remaining to perform sufficient garbage collection work to reclaim memory before the free memory is exhausted.

A realtime application without allocation bursts and sufficient idle time should therefor run with the maximum amount of reserved memory set to `0%`.

The priority default of the memory reservation thread is the Java priority `1` with the scheduler instructed to give preference to other Java threads that run at priority `1` (i.e., with a priority micro adjustment of `-1`). The priority can be changed by setting the Java property `jamaica.reservation_thread_priority` to an integer value larger than or equal to `0`. If set, the memory reservation thread will run at the given Java priority. A value of `0` will result at a Java priority `1` with micro adjustment `-1`, i.e., the scheduler will give preference to other threads running at priority `1`.

7.1.5 Using a GC thread

In JamaicaVM, the garbage collection work is by default performed in the application threads, so there is no need for a dedicated garbage collection thread. However, in an application that provides idle CPU time, one might wish to use this idle time to take load from the main threads and perform garbage collection work during idle time. JamaicaVM permits this by enabling the use of a garbage collection thread (GC thread).

The GC thread is by default not activated. It can be activated by setting a Java system property `jamaica.gcthread_pri`. The value of this property must be the desired thread priority the GC thread should run at. Typically, the lowest Java thread priority `1` is the best value to use an application's idle time.

Since the application may run other Java threads at priority `1`, the property may be set to `0`, which results in a GC thread Java priority `1` and the scheduler set to give preference to other Java threads running at priority `1`.

The GC thread uses this idle time to perform garbage collection work so that the amount of free memory is larger and the application threads can on average perform allocations faster. However, additional CPU time is taken from any other applications on the system that may run at lower priorities.

Even when a GC thread is used, not all of the available CPU time is necessarily used by the Java application. The GC thread will periodically stop its activity when only a little memory was reclaimed during a GC cycle. Lower priority threads may therefore still obtain some CPU time even if a GC thread is used.

7.1.6 Stop-the-world Garbage Collection

For applications that do not have any realtime constraints, but that require the best average time performance, JamaicaVM's builder provides options to disable realtime garbage collection, and to use a stop-the-world garbage collector instead.

In stop-the-world mode, no garbage collection work will be performed until the system runs out of free memory. Then, all threads that perform memory allocation will be stopped to perform garbage collection work until a complete garbage collection cycle is finished and memory was reclaimed. Any thread that does not perform memory allocation may, however, continue execution even while the stop-the-world garbage collector is running.

The builder option `-stopTheWorldGC` enables the stop-the-world garbage collector. Alternatively, the builder option `-constGCwork=-1` may be used, or `-constGCworkFromEnv=`*var* with the environment variable *var* set to `-1`.

JamaicaVM additionally provides an atomic garbage collector that requires stopping of all threads of the Java application during a stop-the-world garbage collection cycle. This has the disadvantage that even threads that do not allocate heap memory will have to be stopped during the GC cycle. However, it avoids the need to track heap modifications performed by threads running parallel to the garbage collector (so called write-barrier code). The result is a slightly increased performance of compiled code.

Specifying the builder option `-atomicGC` enables the atomic garbage collector. Alternatively, the builder option `-constGCwork=-2` may be used, or specify `-constGCworkFromEnv=`*var* with the environment variable *var* set to `-2`.

Please note the use of the memory reservation thread or the GC thread should be disabled when stop-the-world or atomic GC is used.

7.1.7 Recommendations

In summary, to obtain the best performance in your soft-realtime application, follow the following recommendations.

- Set initial heap size as large as possible.
- Set initial heap size and maximum heap size to the same value if possible.
- Set the finalizer thread priority to a low value if your system has enough idle time.
- If your application uses allocation bursts with sufficient CPU idle time in between two allocation bursts, set the amount of reserved memory to fit with the largest allocation burst.
- If your application does not have idle time with intermittent allocation bursts, set the amount of reserved memory to `0%`.

- Enable the GC thread if your system has idle time that can be used for garbage collection.

7.2 Configuration for hard-realtime applications

For predictable execution of memory allocation, more care is needed when selecting memory related options. No dynamic heap size increments should be used if the break introduced by the heap size expansion can harm the realtime guarantees required by the application. Also, the heap size must be set such that the implied garbage collection work is tolerable.

The memory analyzer tool is used to determine the garbage collector settings during a runtime measurement. Together with the `-showNumberOfBlocks` command line option of the Builder tool, they permit an accurate prediction of the time required for each memory allocation. The following sections explain the required configuration of the system.

7.2.1 Usage of the Memory Analyzer tool

The Memory Analyzer is a tool for fine tuning an application's memory requirements and the realtime guarantees that can be given when allocating objects within Java code running on the Jamaica Virtual Machine.

The Memory Analyzer is integrated into the Builder tool. It can be activated by setting the command line option `-analyze=`*accuracy*.

Using the Memory Analyzer Tool is a three-step process: First, an application is built using the Memory Analyzer. The resulting executable file can then be executed to determine its memory requirements. Finally, the result of the execution can be used to fine tune the final version of the application.

7.2.2 Measuring an application's memory requirements

As an example, we will build the HelloWorld example application that was presented in § 2.4. By providing the option `-analyze` to the builder and giving the required accuracy of the analysis in percent, the built application will run in analysis mode to the specified accuracy. In this example, we use an accuracy of 5%:

```
> jamaicabuilder -cp classes -analyze=5 HelloWorld
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
```

```
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__Vcddd02f1f343f157__.c
 [...]
 + tmp/HelloWorld__.c
 + tmp/HelloWorld__.h
 * C compiling 'tmp/HelloWorld__.c'
 [...]
 + tmp/HelloWorld__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                       initial               max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                256MB
GC data:                128KB                 16MB
TOTAL:                 3328KB                343MB
```

The build process is performed exactly as it would be without the `-analyze` option, except that the garbage collector is told to measure the application's memory usage with the given accuracy. The result of this measurement is printed to the console after execution of the application:

```
> ./HelloWorld
              Hello        World!
           Hello       World!
        Hello     World!
     Hello    World!
   Hello   World!
 Hello  World!
 [...]

### Recommended heap size: 4110K (contiguous memory).
### Application used at most 2487232 bytes for reachable objects
on the Java heap
### (accuracy 5%).
###
### Reserved memory is set to 10%.  To obtain lower memory bounds
### or worst-case GC overhead, set reserved memory to 0.
###
### Worst case allocation overhead:
###     heapSize        dynamic GC      const GC work
###     12852K          6               3
###     9874K           7               4
###     8206K           8               4
###     7165K           9               4
###     6443K           10              4
###     5521K           12              5
```

```
###     4968K           14              5
###     4583K           16              6
###     4315K           18              6
###     4110K           20              7
###     3826K           24              8
###     3637K           28              9
###     3505K           32              10
###     3402K           36              11
###     3323K           40              12
###     3209K           48              14
###     3131K           56              17
###     3075K           64              19
###     2945K           96              27
###     2882K           128             36
###     2818K           192             53
###     2789K           256             69
###     2761K           384             100
```

The output consists of the maximum heap memory demand plus a table of possible heap sizes and their allocation overheads for both dynamic and constant garbage collection work. We first consider dynamic garbage collection work, since this is the default.

In this example, the application uses a maximum of 2487232 bytes of memory for the Java heap. The specified accuracy of 5% means that the actual memory usage of the application will be up to 5% less than the measured value, but not higher. JamaicaVM uses the Java heap to store all dynamic data structures internal to the virtual machine (as Java stacks, classes, etc.), which explains the relatively high memory demand for this small application.

7.2.3 Fine tuning the final executable application

In addition to printing the measured memory requirements of the application, in analyze mode Jamaica also prints a table of possible heap sizes and corresponding worst case allocation overheads. The worst case allocation overhead is given in units of garbage collection work that are needed to allocate one block of memory (typically 32 bytes). The amount of time in which these units of garbage collection work can be done is platform dependent. For example, on the PowerPC processor, a unit corresponds to the execution of about 160 machine instructions.

From this table, we can choose the minimum heap size that corresponds to the desired worst case execution time for the allocation of one block of memory. A heap size of 4110K corresponds to a worst case of 20 units of garbage collection work (3200 machine instructions on the PowerPC) per block allocation, while a smaller heap size of, for example, 3323K can only guarantee a worst case

execution time of 40 units of garbage collection work (that is, 6400 PowerPC-instructions) per block allocation.

If we find that for our application 14 units of garbage collection work per allocation is sufficient to satisfy all realtime requirements, we can build the final application using a heap of 4968K:

```
> jamaicabuilder -cp classes -heapSize=4968K -maxHeapSize=4968K \
>   HelloWorld
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__Vcddd02f1f343f157__.c
 [...]
 + tmp/HelloWorld__.c
 + tmp/HelloWorld__.h
 * C compiling 'tmp/HelloWorld__.c'
 [...]
 + tmp/HelloWorld__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                       initial                max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             4968KB                4968KB
GC data:                310KB                 310KB
TOTAL:                 6430KB                  77MB
```

Note that both options, `heapSize` and `maxHeapSize`, are set to the same value. This creates an application that has the same initial heap size and maximum heap size, i.e., the heap size is not increased dynamically. This is required to ensure that the maximum of 14 units of garbage collection work per unit of allocation is respected during the whole execution of the application. With a dynamically growing heap size, an allocation that happens to require increasing the heap size will otherwise be blocked until the heap size is increased sufficiently.

The resulting application will now run with the minimum amount of memory that guarantees the selected worst case execution time for memory allocation. The actual amount of garbage collection work that is performed is determined dynamically depending on the current state of the application (including, for example, its memory usage) and will in most cases be significantly lower than the described worst case behavior, so that on average an allocation is significantly cheaper than the worst case allocation cost.

7.2.4 Constant Garbage Collection Work

For applications that require best worst case execution times, where average case execution time is less important, Jamaica also provides the option to statically select the amount of garbage collection work. This forces the given amount of garbage collection work to be performed at any allocation, without regard to the current state of the application. The advantage of this static mode is that worst case execution times are lower than using dynamic determination of garbage collection work. The disadvantage is that any allocation requires this worst case amount of garbage collection work.

The output generated using the option `-analyze` also shows possible values for the constant garbage collection option. A unit of garbage collection work is the same as in the dynamic case — about 160 machine instructions on the PowerPC processor.

Similarly, if we want to give the same guarantee of 14 units of work for the worst case execution time of the allocation of a block of memory with constant garbage collection work, a heap size of 3209K bytes is sufficient. To inform the builder that constant garbage collection work should be used, the option `-constGCwork` and the number of units of work should be specified when building the application:

```
> jamaicabuilder -cp classes -heapSize=3209K -maxHeapSize=3209K \
>   -constGCwork=14 HelloWorld
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__Vcddd02f1f343f157__.c
 [...]
 + tmp/HelloWorld__.c
 + tmp/HelloWorld__.h
 * C compiling 'tmp/HelloWorld__.c'
 [...]
 + tmp/HelloWorld__nc.o
 * linking
 * stripping
Application memory demand will be as follows:
                      initial                max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             3209KB                3209KB
GC data:                200KB                 200KB
TOTAL:                 4561KB                  75MB
```

7.2.5 Comparing dynamic mode and constant GC work mode

Which option you should choose (dynamic mode or constant garbage collection) depends strongly on the kind of application. If worst case execution time and low jitter are the most important criteria, constant garbage collection work will usually provide the better performance with smaller heap sizes. But if average case execution time is also an issue, dynamic mode will typically give better overall throughput, even though for equal heap sizes the guaranteed worst case execution time is longer with dynamic mode than with constant garbage collection work.

Gradual degradation may also be important. Dynamic mode and constant garbage collection work differ significantly when the application does not stay within the memory bounds that were fixed when the application was built.

There are a number of reasons an application might be using more memory:

- The application input data might be bigger than originally anticipated.
- The application was built with an incorrect or outdated `-heapSize` argument.
- A bug in the application may be causing a memory leak and gradual use of more memory than expected.

Whatever the reason, it may be important in some environments to understand the behavior of memory management in the case the application exceeds the assumed heap usage.

In dynamic mode, the worst-case execution time for an allocation can no longer be guaranteed as soon as the application uses more memory. But as long as the excess heap used stays small, the worst-case execution time will increase only slightly. This means that the original worst-case execution time may not be exceeded at all or only by a small amount. However, the garbage collector will still work properly and recycle enough memory to keep the application running.

If the constant garbage collection work option is chosen, the amount of garbage-collection work will not increase even if the application uses more memory than originally anticipated. Allocations will still be made within the same worst-case execution time. Instead, the collector cannot give a guarantee that it will recycle memory fast enough. This means that the application may fail abruptly with an out-of-memory error. Static mode does not provide graceful degradation of performance in this case, but may cause abrupt failure even if the application exceeds the expected memory requirements only slightly.

7.2.6 Determination of the worst case execution time of an allocation

As we have just seen, the worst case execution time of an allocation depends on the amount of garbage collection work that has to be performed for the allocation. The configuration of the heap as shown above gives a worst case number of garbage collection work units that need to be performed for the allocation of one block of memory. In order to determine the actual time an allocation might take in the worst case, it is also necessary to know the number of blocks that will be allocated and the platform dependent worst case execution time of one unit of garbage collection work.

For an allocation statement S we get the following equation to calculate the worst case-execution time:

$$\text{wcet}(S) = \text{numblocks}(S) \cdot \text{max-gc-units} \cdot \text{wcet-of-gc-unit}$$

Where

- wcet(S) is the worst case execution time of the allocation
- numblocks(S) gives the number of blocks that need to be allocated
- max-gc-units is the maximum number of garbage collection units that need to be performed for the allocation of one block
- wcet-of-gc-unit is the platform dependent worst case execution time of a single unit of garbage collection work.

7.2.7 Examples

Imagine that we want to determine the worst-case execution time (wcet) of an allocation of a StringBuffer object, as was done in the HelloWorld.java example shown above. If this example was built with the dynamic garbage collection option and a heap size of 443K bytes, we get

$$\text{max-gc-units} = 14$$

as has been shown above. If our target platform gives a worst case execution time for one unit of garbage collection work of $1.6\mu s$, we have

$$\text{wcet-of-gc-unit} = 1.6\mu s$$

We use the `-showNumberOfBlocks` command line option to find the number of blocks required for the allocation of a java.lang.StringBuffer object. Actually this option shows the number of blocks for all classes used by the application even when for this example we are only interested in the mentioned class.

```
> jamaicabuilder -cp classes -showNumberOfBlocks HelloWorld

[...]
java/lang/String$CIO                                    1
java/lang/String$GetBytesCacheEntry                     1
java/lang/String$WeakSet                                1
java/lang/StringBuffer                                  2
java/lang/StringBuilder                                 2
java/lang/StringCoding                                  1
java/lang/StringCoding$1                                1
java/lang/StringCoding$StringDecoder                    1
[...]
```

A StringBuffer object requires two blocks of memory, so that

$$\text{numblocks}(\texttt{new StringBuffer()}) = 2$$

and the total worst case-execution time of the allocation becomes

$$\text{wcet}(\texttt{new StringBuffer()}) = 2 \cdot 14 \cdot 1.6\mu s = 44.8\mu s$$

Had we used the constant garbage collection option with the same heap size, the amount of garbage collection work on an allocation of one block could have been fixed at 6 units. In that case the worst case execution time of the allocation becomes

$$\text{wcet}_{\text{constGCwork}}(\texttt{new StringBuffer()}) = 2 \cdot 6 \cdot 1.6\mu s = 19.2\mu s$$

Chapter 8

Debugging Support

Jamaica supports the debugging facilities of integrated development environments (IDEs) such as Eclipse and Netbeans. These are popular IDEs for the Java platform. Debugging is possible on instances of the JamaicaVM running on the host platform, as well as for applications built with Jamaica, which run on an embedded device. The latter requires that the device provides network access.

In this chapter, it is shown how to set up the IDE debugging facilities with Jamaica. A reference section towards the end briefly explains the underlying technology (JPDA) and the supported options.

8.1 Enabling the Debugger Agent

While debugging the IDE's debugger needs to connect to the virtual machine or the running application in order to inspect the VM's state, set breakpoints, start and stop execution and so forth. Jamaica contains a communication agent, which must be either enabled (for the VM) or built into the application. This is done through the `agentlib` option.

```
> jamaicavm -agentlib:BuiltInAgent=transport=dt_socket, \
>   address=localhost:4000,server=y,suspend=y HelloWorld
```

launches JamaicaVM with debug support enabled and `HelloWorld` as the main class. The VM listens on port 4000 at `localhost`. The VM is suspended and waits for the debugger to connect. It then executes normally until a breakpoint is reached.

In order to build debugging support into an application, the builder option `-agentlib=BuiltInAgent...` should be used. If the application is to be debugged on an (embedded) device, `localhost` must be replaced by the network address of the device.

8.2 Configuring the IDE to Connect to Jamaica

Before being able to debug a project, the code needs to compile and basically run. Before starting a debugging session, the debugger must be configured to connect to the VM by specifying the VM's host address and port. Normally, this is done by setting up a *debug configuration*.

In Eclipse 3.5, for example, select the menu item

```
Run > Debug Configurations....
```

In the list of available items presented on the left side of the dialog window (see Fig. 8.1), choose a new configuration for a remote Java application, then

- configure the debugger to connect to the VM by choosing connection type *socket attach* and
- enter the VM's network address and port as the connection properties *host* and *port*.

Clicking on `Debug` attaches the debugger to the VM and starts the debugging session. If the VM's communication agent is set to suspending the VM before loading the main class, the application will only run after instructed to do so through the debugger via commands from the `Run` menu. In Eclipse, breakpoints may be set conveniently by double-clicking in the left margin of the source code.

For instructions on debugging, the documentation of the used debugger should be consulted — in Eclipse, for example, though the `Help` menu.

The Jamaica Eclipse Plug-In (see § 4) provides the required setup for debugging with the JamaicaVM on the host system automatically. It is sufficient to select Jamaica as the Java Runtime Environment of the project.

8.3 Reference Information

Jamaica supports the Java Platform Debugger Architecture (JPDA). Debugging is possible with IDEs that support the JPDA. Tab. 8.1 shows the debugging options accepted by Jamaica's communication agent. The Jamaica Debugging Interface has the following limitations:

- Local variables of compiled methods cannot be examined
- Stepping through a compiled method is not supported
- Setting a breakpoint in a compiled method will silently be ignored
- Notification on field access/modification is not available

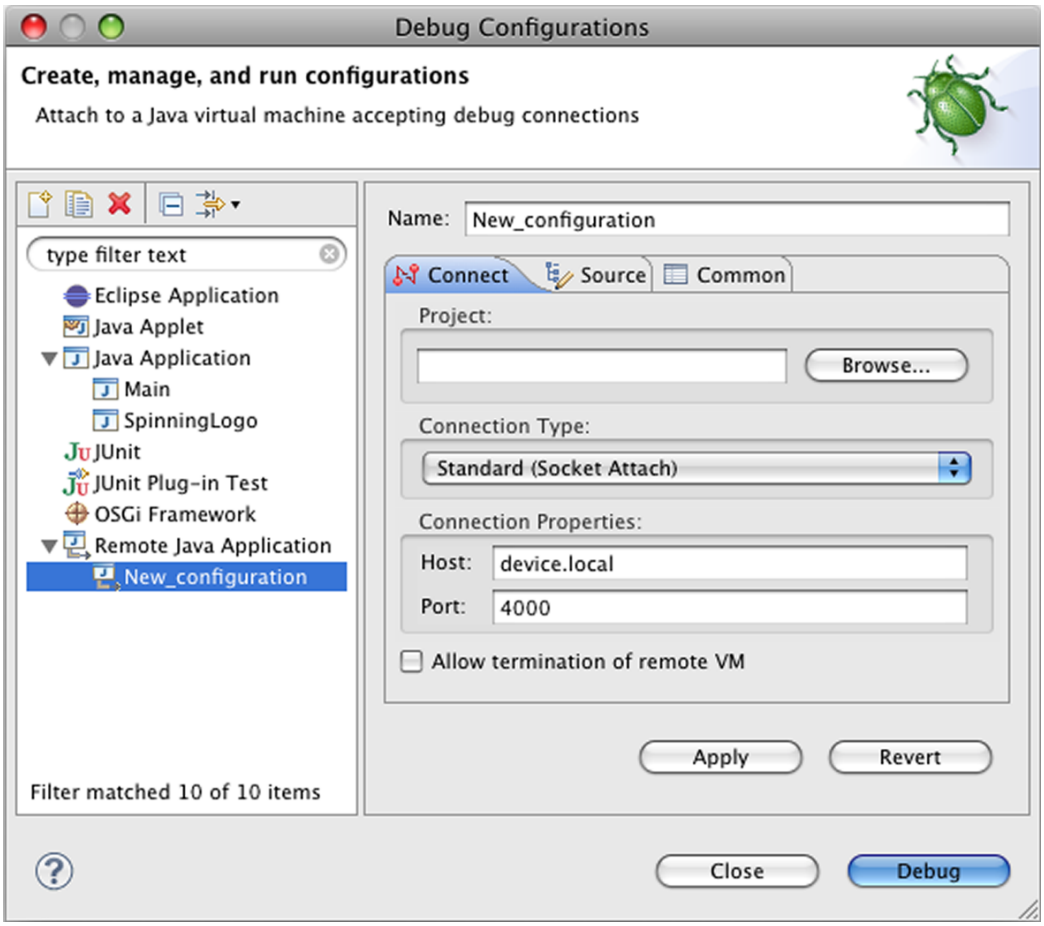


Figure 8.1: Setting up a remote debugging connection in Eclipse 3.5

Syntax	Description
`transport=dt_socket`	The only supported transport protocol is `dt_socket`.
`address=[`*host*`:]`*port*	Transport address for the connection.
`server=y|n`	If `y`, listen for a debugger application to attach; otherwise, attach to the debugger application at the specified address.
`suspend=y|n`	If `y`, suspend this VM until connected to the debugger.

Table 8.1: Arguments of Jamaica's communication agent

- Information about java monitors cannot be retrieved

The Java Platform Debugger Architecture (JPDA) consists of three interfaces designed for use by debuggers in development environments for desktop systems. The Java Virtual Machine Tools Interface (JVMTI) defines the services a VM must provide for debugging.[1] The Java Debug Wire Protocol (JDWP) defines the format of information and requests transferred between the process being debugged and the debugger front end, which implements the Java Debug Interface (JDI). The Java Debug Interface defines information and requests at the user code level.

A JPDA Transport is a method of communication between a debugger and the virtual machine that is being debugged. The communication is connection oriented — one side acts as a server, listening for a connection. The other side acts as a client and connects to the server. JPDA allows either the debugger application or the target VM to act as the server. The transport implementations of Jamaica allows communications between processes running on different machines.

[1]The JVMTI is a replacement for the Java Virtual Machine Debug Interface (JVMDI) which has been deprecated.

Chapter 9

The Real-Time Specification for Java

JamaicaVM supports the Real-Time Specification for Java V1.0.2 (RTSJ), see [2]. The specification is available at `http://www.rtsj.org`. The API documentation of the JamaicaVM implementation is available online at `http://www.aicas.com/jamaica/doc/rtsj_api/` and is included in the API documentation of the Jamaica class library:

jamaica-home`/doc/jamaica_api/index.html`.

The RTSJ resides in package `javax.realtime`. It is generally recommended that you refer to the RTSJ documentation provided by aicas since it contains a detailed description of the behavior of the RTSJ functions and includes specific comments on the behavior of JamaicaVM at places left open by the specification.

9.1 Realtime programming with the RTSJ

The aim of the Real-Time Specification for Java (RTSJ) is to extend the Java language definition and the Java standard libraries to support realtime threads, i.e., threads whose execution conforms to certain timing constraints. Nevertheless, the specification is compatible with different Java environments and backwards compatible with existing non-realtime Java applications.

The most important improvements of the RTSJ affect the following seven areas:

- thread scheduling,
- memory management,

- synchronization,
- asynchronous events,
- asynchronous flow of control,
- thread termination, and
- physical memory access.

With this, the RTSJ also covers areas that are not directly related to realtime applications. However, these areas are of great importance to many embedded realtime applications such as direct access to physical memory (e.g., memory mapped I/O) or asynchronous mechanisms.

9.1.1 Thread Scheduling

To enable the development of realtime software in an environment with a garbage collector that stops the execution of application threads in an unpredictable way, new thread classes `RealtimeThread` and `NoHeapRealtimeThread` are defined. These thread types are unaffected or at least less heavily affected by garbage collection activity. Also, at least 28 new priority levels, logically higher than the priority of the garbage collector, are available for these threads.

9.1.2 Memory Management

In order for realtime threads not to be affected by garbage collector activity, they need to use memory areas that are not under the control of the garbage collector. New memory classes, `ImmortalMemory` and `ScopedMemory`, provide these memory areas. One important consequence of the use of special memory areas is, of course, that the advantages of dynamic memory management are not fully available to realtime threads.

9.1.3 Synchronization

In realtime systems with threads of different priority levels, priority inversion situations must be avoided. Priority inversion occurs when a thread of high priority is blocked by waiting for a monitor that is owned by a thread of a lower priority. The RTSJ provides the alternatives priority inheritance and the priority ceiling protocol to avoid priority inversion.

9.1.4 Example

The RTSJ offers powerful features that enable the development of realtime applications. The following program in shows an example how the RTSJ can be used in practice.

```
/**
 * Small demonstration of a periodic thread in Java:
 */

import javax.realtime.*;
public class HelloRT
{
  public static void main(String [] args)
  {
    /* priority for new thread: min+10 */
    int pri =
      PriorityScheduler.instance().getMinPriority() + 10;
    PriorityParameters prip = new PriorityParameters(pri);

    /* period: 20ms */
    RelativeTime period =
      new RelativeTime(20 /* ms */, 0 /* ns */);

    /* release parameters for periodic thread: */
    PeriodicParameters perp =
     new PeriodicParameters(null, period, null, null, null, null);

    /* create periodic thread: */
   RealtimeThread rt = new RealtimeThread(prip, perp)
      {
        public void run()
        {
          int n=1;
          while (waitForNextPeriod() && (n<100))
            {
              System.out.println("Hello "+n);
              n++;
            }
        }
      };

    /* start periodic thread: */
    rt.start();
  }
}
```

In this example, a periodic thread is created. This thread becomes active every 20ms and writes output onto the standard console. A `RealtimeThread` is used

to implement this task. The priority and the length of the period of this periodic thread need to be provided. A call to `waitForNextPeriod()` causes the thread to wait after the completion of one activation for the start of the next period. An introduction to the RTSJ with numerous further examples is given in the book by Peter Dibble [3].

The RTSJ provides a solution for realtime programming, but it also brings new difficulties to the developer. The most important consequence is that applications have to be split strictly into two parts: a realtime and a non-realtime part. The communication between these parts is heavily restricted: realtime threads cannot perform memory operations such as the allocation of objects on the normal heap which is under the control of the garbage collector. Synchronization between realtime and non-realtime threads is heavily restricted since it can cause realtime threads to be blocked by the garbage collector.

9.2 Realtime Garbage Collection

In JamaicaVM, a system that supports realtime garbage collection, this strict separation into realtime and non-realtime threads is not necessary. The strict splitting of an application is consequently not required. Threads are activated depending only on their priorities.

The realtime garbage collector performs its work predictably within the application threads. It is activated when memory is allocated. The work done on an allocation must be preemptible, so that more urgent threads can become active.

The implementation of a realtime garbage collector must solve a number of technical challenges. Garbage collector activity must be performed in very small single increments of work. In JamaicaVM, one increment consists of garbage collecting only 32 bytes of memory. On every allocation, the allocating thread "pays" for the memory by performing a small number of these increments. The number of increments can be analyzed, such that this is possible even in realtime code.

The RTSJ provides a powerful extension to the Java specification. Its full power, however, is achieved only by the combination with a realtime garbage collector that helps to overcome its restrictions.

9.3 Relaxations in JamaicaVM

Because JamaicaVM uses a realtime garbage collector, the limitations that the Real-Time Specification for Java imposes on realtime programming are not imposed on realtime applications developed for JamaicaVM. The limitations that are

relaxed in JamaicaVM affect the use of memory areas, thread priorities, runtime checks and static initializers.

For the development of applications that do not make use of these relaxations, the builder option `strictRTSJ` (see below) can be set to disable these relaxations.

9.3.1 Use of Memory Areas

Because JamaicaVM's realtime garbage collector does not interrupt application threads, it is unnecessary for objects of class `RealtimeThread` or even of `NoHeapRealtimeThread` to run in their own memory area not under the control of the garbage collector. Instead, any thread can use and access the normal garbage collected heap.

Nevertheless, any thread can make use of the new memory areas such as `LTMemory` or `ImmortalMemory` if the application developer wishes to do so. Since these memory classes are not controlled by the garbage collector, allocations do not require garbage collector activity and may be faster or more predictable than allocations on the normal heap. However, great care is required in these memory areas to avoid memory leaks, since temporary objects allocated in scoped or immortal memory will not be reclaimed automatically.

9.3.2 Thread Priorities

In JamaicaVM, `RealtimeThread`, `NoHeapRealtimeThread` and normal `Thread` objects all share the same priority range. The lowest possible thread priority for all of these threads is `MIN_PRIORITY` which is defined in package `java.lang`, class `Thread`. The the highest possible priority may be obtained by querying `instance().getMaxPriority()` in package `javax.realtime`, class `PriorityScheduler`.

9.3.3 Runtime checks for NoHeapRealtimeThread

Even `NoHeapRealtimeThread` objects will be exempt from interruption by garbage collector activities. JamaicaVM does not, therefore, prevent these threads from accessing objects allocated on the normal heap. Runtime checks that typically ensure that these threads do not access objects allocated on the heap are not performed by JamaicaVM.

9.3.4 Static Initializers

To permit the initialization of classes even if their first reference is performed within `ScopedMemory` or `ImmortalMemory` within a `RealtimeThread` or `NoHeapRealtimeThread`, and to permit the access of static fields such as `System.out` from within these threads, static initializers are typically executed within `ImmortalMemory` that is accessible by all threads. However, this prevents these objects from being reclaimed when they are no longer used. Also, it can cause a serious memory leak if dynamic class loading is used since memory allocated by the static initializers of dynamically loaded classes will never be reclaimed.

Since JamaicaVM does not limit access to heap objects within any threads, there is no need to execute static initializers within ImmortalMemory. However, objects allocated in static initializers typically must be accessible by all threads, so they cannot be allocated in a scoped memory area if this happens to be the current thread's allocation environment when the static initializer is executed.

JamaicaVM therefore executes all static initializers within heap memory. Objects allocated by static initializers may be accessed by all threads, and they may be reclaimed by the garbage collector. There is no memory leak if classes are loaded dynamically by a user class loader.

9.3.5 Class PhysicalMemoryManager

According to the RTSJ, names and instances of class `PhysicalMemoryTypeFilter` in package `javax.realtime` that are passed to method `registerFilter` of class `PhysicalMemoryManager` in the same package must be allocated in immortal memory. This requirement does not exist in JamaicaVM.

9.4 Strict RTSJ Semantics

When the builder option `strictRTSJ` is chosen, the relaxations just described are deactivated and strict RTSJ semantics are enforced. Applications that should be portable to different RTSJ implementations should consequently be developed with this options switched on when an application is built.

9.4.1 Use of Memory Areas

All NoHeapRealtimeThreads of applications built in `strictRTSJ` mode must run in scoped or immortal memory. In addition, the thread object itself, the thread logic, scheduling, release, memory and group parameters must not be allocated

in heap memory. Otherwise, a MemoryAccessError is thrown on the creation of such a thread.

RealtimeThreads are free to use heap memory even in `strictRTSJ` mode, and they still profit from the lack of garbage collection pauses in JamaicaVM. Application code that needs to be portable to other Java implementations that are not based on realtime garbage collection should not use heap memory for time critical threads.

Normal threads are not allowed to enter non-heap memory areas in `strict-RTSJ` mode.

9.4.2 Thread priorities

Thread priorities for normal Java threads must be in a range specified in package `java.lang` class `Thread`. The minimal priority is `MIN_PRIORITY`, the maximal priority `MAX_PRIORITY`.

`RealtimeThreads` and `NoHeapRealtimeThreads` share the priority range defined in `javax.realtime`, class `PriorityScheduler`, and which may be obtained by querying method `instance().getMinPriority()` and `instance().getMaxPriority()`.

9.4.3 Runtime checks for NoHeapRealtimeThread

If `strictRTSJ` is set, runtime checks on all memory read operations (i.e., accesses to static and instance fields and accesses to reference array elements) are checked to ensure that no object on the garbage collected heap is touched by a NoHeapRealtimeThread.

These runtime checks are required by classical Java implementations with a non-realtime garbage collector. They may impose an important runtime overhead on the application.

9.4.4 Static Initializers

When `strictRTSJ` is set, static initializers are executed within immortal memory. This means that all objects allocated by static initializers are accessible by all threads. Care is required since any allocations performed within static initializers of classes that are loaded dynamically into a system will never be recycled. Dynamic class loading consequently poses a severe risk of introducing a memory leak into the system.

9.4.5 Class PhysicalMemoryManager

When `strictRTSJ` is set, names and instances of class `PhysicalMemoryTypeFilter` in package `javax.realtime` that are passed to method `registerFilter` of class `PhysicalMemoryManager` in the same package must be allocated in immortal memory as required by the RTSJ.

9.5 Limitations of RTSJ Implementation

The following methods or classes of the RTSJ are not fully supported in JamaicaVM 6.2:

- Class `VTPhysicalMemory`
- Class `LTPhysicalMemory`
- Class `ImmortalPhysicalMemory`
- In class `AsynchronouslyInterruptedException` the deprecated method `propagate()` is not supported.
- The class `AffinitySet` is currently supported for `Threads` and `BoundAsyncEventHandlers` only, but not for the class `ProcessingGroupParameters`. The default sets supported by Jamaica are sets with either exactly one single element or the set of all CPUs. The CPU ids used on the Java side are 0 though $n - 1$ when n CPUs are used, while the values provided to the -Xcpus builder argument are the CPU ids used by the underlying OS.

Cost monitoring is supported and cost overrun handlers will be fired on a cost overrun. However, cost enforcement is currently not supported. The reason is that stopping a thread or handler that holds a lock is dangerous since it might cause a deadlock. RTSJ cost enforcement is based on the CPU cycle counter. This is available on x86 and PPC systems only, so cost enforcement will not work on other systems.

Chapter 10

Guidelines for Realtime Programming in Java

10.1 General

Since the timeliness of realtime systems is just as important as their functional correctness, realtime Java programmers must take more care using Java than other Java users. In fact, realtime Java implementations in general and the JamaicaVM in particular offer a host of features not present in standard Java implementations.

The JamaicaVM offers a myriad of sometimes overlapping features for realtime Java development. The realtime Java developer needs to understand these features and when to apply them. Particularly, with realtime specific features pertaining to memory management and task interaction, the programmer needs to understand the trade-offs involved. This chapter does not offer cut and dried solutions to specific application problems, but instead offers guidelines for helping the developer make the correct choice.

10.2 Computational Transparency

In contrast to normal software development, the development of realtime code requires not only the correctness of the code, but also the timely execution of the code. For the developer, this means that not only the result of each statement is important, but also the approximate time required to perform the statement must be obvious. One need not know the exact execution time of each statement when this statement is written, as the exact determination of the worst case execution time can be performed by a later step; however, one should have a good understanding of the order of magnitude in time a given code section needs for execution early on in the coding process. For this, the computational complexity can be described in

categories such as a few machine cycles, a few hundred machine cycles, thousands of machine cycles or millions of machine cycles. Side effects such as blocking for I/O operations or memory allocation should be understood as well.

The term *computational transparency* refers to the degree to which the computational effort of a code sequence written in a programming language is obvious to the developer. The closer a sequence of commands is to the underlying machine, the more transparent that sequence is. Modern software development tries to raise the abstraction level at which programmers ply their craft. This tends to reduce the cost of software development and increase its robustness. Often however, it masks the real work the underlying machine has to do, thus reducing the computational transparency of code.

Languages like Assembler are typically completely computationally transparent. The computational effort for each instruction can be derived in a straightforward way (e.g., by consulting a table of instruction latency rules). The range of possible execution times of different instructions is usually limited as well. Only very few instructions in advanced processor architectures have an execution time of more than `O(1)`.

Compiled languages vary widely in their computational complexity. Programming languages such as C come very close to full computational transparency. All basic statements are translated into short sequences of machine code instructions. More abstract languages can be very different in this respect. Some simple constructs may operate on large data structures, e.g., sets, thus take an unbounded amount of time.

Originally, Java was a language that was very close to C in its syntax with comparable computational complexity of its statements. Only a few exceptions were made. Java has evolved, particularly in the area of class libraries, to ease the job of programming complex systems, at the cost of diminished computational transparency. Therefore a short tour of the different Java statements and expressions, noting where a non-obvious amount of computational effort is required to perform these statements with the Java implementation JamaicaVM, is provided here.

10.2.1 Efficient Java Statements

First the good news. Most Java statements and expressions can be implemented in a very short sequence of machine instructions. Only statements or constructs for which this is not so obvious are considered further.

10.2.1.1 Dynamic Binding for Virtual Method Calls

Since Java is an object-oriented language, dynamic binding is quite common. In the JamaicaVM dynamic binding of Java methods is performed by a simple lookup in the method table of the class of the target object. This lookup can be performed with a small and constant number of memory accesses. The total overhead of a dynamically bound method invocation is consequently only slightly higher than that of a procedure call in a language like C.

10.2.1.2 Dynamic Binding for Interface Method Calls

Whereas single inheritance makes normal method calls easy to implement efficiently, calling methods via an interface is more challenging. The multiple inheritance implicit in Java interfaces means that a simple dispatch table as used by normal methods can not be used. In the JamaicaVM the time needed to find the called method is linear with the number of interfaces implemented by the class.

10.2.1.3 Type Casts and Checks

The use of type casts and type checks is very frequent in Java. One example is the following code sequence that uses an `instanceof` check and a type cast:

```
...
Object o = vector.elementAt(index);

if (o instanceof Integer)
  sum = sum + ((Integer)o).intValue();
...
```

These type checks also occur implicitly whenever a reference is stored in an array of references to make sure that the stored reference is compatible with the actual type of the array. Type casts and type checks within the JamaicaVM are performed in constant time with a small and constant number of memory accesses. In particular, `instanceof` is more efficient than method invocation.

10.2.1.4 Generics (JDK 1.5)

The generic types (*generics*) introduced in JDK 1.5 avoid explicit type cases that are required using abstract data types with older versions of Java. Using generics, the type cast in this code sequence

```
ArrayList list = new ArrayList();
list.add(0, "some string");
String str = (String) list.get(0);
```

is no longer needed. The code can be written using a generic instance of `ArrayList` that can only hold strings as follows.

```
ArrayList<String> list =  new ArrayList<String>();
list.add(0, "some string");
String str = list.get(0);
```

Generics still require type casts, but these casts are hidden from the developer. This means that access to `list` using `list.get(0)` in this example in fact performs the type cast to `String` implicitly causing additional runtime overhead. However, since type casts are performed efficiently and in constant time in JamaicaVM, the use of generics can be recommended even in time-critical code wherever this appears reasonable for a good system design.

10.2.2 Non-Obvious Slightly Inefficient Constructs

A few constructs have some hidden inefficiencies, but can still be executed within a short sequence of machine instructions.

10.2.2.1 `final` Local Variables

The use of `final` local variables is very tempting in conjunction with anonymous inner classes since only variables that are declared `final` can be accessed from code in an anonymous inner class. An example for such an access is shown in the following code snippet:

```
final int data = getData();

new RealtimeThread(new PriorityParameters(pri))
  {
    public void run()
      {
        for (...)
          {
            ...
            x = data;
            ...
          }
      }
  }
```

All uses of the local variable within the inner class are replaced by accesses to a hidden field. In contrast to normal local variables, each access requires a memory access.

10.2.2.2 Accessing `private` Fields from Inner Classes

As with the use of `final` local variables, any `private` fields that are accessed from within an inner class require the call to a hidden access method since these accesses would otherwise not be permitted by the virtual machine.

10.2.3 Statements Causing Implicit Memory Allocation

Thus far, only execution time has been considered, but memory allocation is also a concern for safety-critical systems. In most cases, memory allocation in Java is performed explicitly by the keyword `new`. However, some statements perform memory allocations implicitly. These memory allocations do not only require additional execution time, but they also require memory. This can be fatal within execution contexts that have limited memory, e.g., code running in a `ScopedMemory` or `ImmortalMemory` as it is required by the Real-Time Specification for Java for `NoHeapRealtimeThreads`. A realtime Java programmer should be familiar with all statements and expressions which cause implicit memory allocation.

10.2.3.1 String Concatenation

Java permits the composition of strings using the plus operator. Unlike adding scalars such as `int` or `float` values, string concatenation requires the allocation of temporary objects and is potentially very expensive.

As an example, the instruction

```
int    x     = ...;
Object thing = ...;

String msg = "x is " + x + " thing is " + thing;
```

will be translated into the following statement sequence:

```
int    x     = ...;
Object thing = ...;

StringBuffer tmp_sb = new StringBuffer();
tmp_sb.append("x is ");
tmp_sb.append(x);
tmp_sb.append(" thing is ");
tmp_sb.append(thing.toString());
String msg = tmp_sb.toString();
```

The code contains hidden allocations of a `StringBuffer` object, of an internal character buffer that will be used within this `StringBuffer`, a temporary string allocated for `thing.toString()`, and the final string returned by `tmp_sb.toString()`.

Apart from these hidden allocations, the hidden call to `thing.toString()` can have an even higher impact on the execution time, since method `toString` can be redefined by the actual class of the instance referred to by `thing` and can cause arbitrarily complex computations.

10.2.3.2 Array Initialization

Java also provides a handy notation for array initialization. For example, an array with the first 8 Fibonacci numbers can be declared as

```
int[] fib = { 1, 1, 2, 3, 5, 8, 13, 21 };
```

Unlike C, where such a declaration is converted into preinitialized data, the Java code performs a dynamic allocation and is equivalent to the following code sequence:

```
int[] fib = new int[8];
fib[0] = 1;
fib[1] = 1;
fib[2] = 2;
fib[3] = 3;
fib[4] = 5;
fib[5] = 8;
fib[6] = 13;
fib[7] = 21;
```

Initializing arrays in this way should be avoided in time critical code. When possible, constant array data should be initialized within the static initializer of the class that uses the data and assigned to a static variable that is marked `final`. Due to the significant code overhead, large arrays should instead be loaded as a resource, using the Java standard API (via method `getResourceAsStream` from class `java.lang.Class`).

10.2.3.3 Autoboxing (JDK 1.5)

Unlike some Scheme implementations, primitive types in Java are not internally distinguishable from pointers. This means that in order to use a primitive data type where an object is needed, the primitive needs to be boxed in its corresponding object. JDK 1.5 introduces autoboxing which automatically creates objects for values of primitive types such as `int`, `long`, or `float` whenever these values are assigned to a compatible reference. This feature is purely syntactic. An expression such as

```
o = new Integer(i);
```

can be written as

```
o = i;
```

Due to the hidden runtime overhead for the memory allocation, autoboxing should be avoided in performance critical code. Within code sequences that have heavy restrictions on memory demand, such as realtime tasks that run in `Immortal-Memory` or `ScopedMemory`, autoboxing should be avoided completely since it may result in hidden memory leaks.

10.2.3.4 For Loop Over Collections (JDK 1.5)

JDK 1.5 also introduces an extended `for` loop. The extension permits the iteration of a `Collection` using a simple for loop. This feature is purely syntactic. A loop such as

```
ArrayList list = new ArrayList();
for (Iterator i = list.iterator(); i.hasNext();)
  {
    Object value = i.next();
    ...
  }
```

can be written as

```
ArrayList list = new ArrayList();
for (Object value : list)
  {
    ...
  }
```

The allocation of a temporary `Iterator` that is performed by the call to `list.iterator()` is hidden in this new syntax.

10.2.3.5 Variable Argument Lists (JDK 1.5)

There is still another feature of JDK 1.5 that requires implicit memory allocation. The new variable argument lists for methods is implemented by an implicit array allocation and initialization. Variable argument lists should consequently be avoided.

10.2.4 Operations Causing Class Initialization

Another area of concern for computational transparency is class initialization. Java uses `static` initializers for the initialization of classes on their first use. The first use is defined as the first access to a static method or static field of the class in question, its first instantiation, or the initialization of any of its subclasses.

The code executed during initialization can perform arbitrarily complex operations. Consequently, any operation that can cause the initialization of a class may take arbitrarily long for its first execution. This is not acceptable for time critical code.

Consequently, the execution of static initializers has to be avoided in time critical code. There are two ways to achieve this: either time critical code must not perform any statements or expressions that may cause the initialization of a class, or the initialization has to be made explicit.

The statements and expressions that cause the initialization of a class are

- reading a static field of another class,
- writing a static field of another class,
- calling a static method of another class, and
- creating an instance of another class using `new`.

An explicit initialization of a class `C` is best performed in the static initializer of the class `D` that refers to `C`. One way to do this is to add the following code to class `D`:

```
/* initialize class C: */
static { C.class.initialize(); }
```

The notation `C.class` itself has its own disadvantages (see § 10.2.5). So, if possible, it may be better to access a static field of the class causing initialization as a side effect instead.

```
/* initialize class C: */
static { int ignore = C.static_field; }
```

10.2.5 Operations Causing Class Loading

Class loading can also occur unexpectedly. A reference to the class object of a given class `C` can be obtained using *classname*`.class` as in the following code:

```
Class class_C = C.class;
```

This seemingly harmless operation is, however, transformed into a code sequence similar to the following code:

```
static Class class$(String name)
  {
    try { return Class.forName(name); }
    catch (ClassNotFoundException e)
      {
        throw new NoClassDefFoundError(e.getMessage());
      }
  }

static Class class$C;

...

Class tmp;
if (class$C == null)
  {
    tmp = class$("C");
    class$C = tmp;
```

```
  }
else
  {
    tmp = class$C;
  }
Class class_C = tmp;
```

This code sequence causes loading of new classes from the current class loading context. I.e., it may involve memory allocation and loading of new class files. If the new classes are provided by a user class loader, this might even involve network activity, etc.

Starting with JDK 1.5, the *classname*`.class` notation will be supported by the JVM directly. The complex code above will be replaced by a simple bytecode instruction that references the desired class directly. Consequently, the referenced class can be loaded by the JamaicaVM at the same time the referencing class is loaded and the statement will be replaced by a constant number of memory accesses.

10.3 Supported Standards

Thus far, only standard Java constructs have been discussed. However libraries and other APIs are also an issue. Timely Java development needs support for timely execution and device access. There are also issues of certifiability to consider. The JamaicaVM has at least some support for all of the following APIs.

10.3.1 Real-Time Specification for Java

The Real-Time Specification for Java (RTSJ) provides functionality needed for time-critical Java applications. RTSJ introduces an additional API of Java classes, mainly with the goal of providing a standardized mechanism for realtime extensions of Java Virtual Machines. RTSJ extensions also cover other areas of great importance to many embedded realtime applications, such as direct access to physical memory (e.g., memory mapped I/O) or asynchronous mechanisms.

Currently, RTSJ is available for the JamaicaVM and for TimeSys JTime. JVMs from other vendors are likely to follow in the future.

10.3.1.1 Thread Scheduling in the RTSJ

Ensuring that Java programs can execute in a timely fashion was a main goal of the RTSJ. To enable the development of realtime software in an environment with a garbage collector that stops the execution of application threads in an unpredictable way (see Fig. 10.1), the new thread classes `RealtimeThread` and

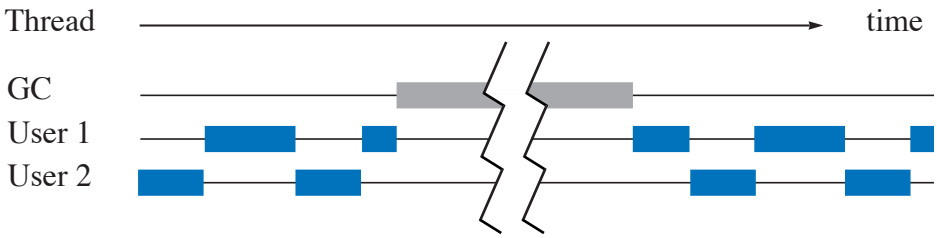


Figure 10.1: Java Threads in a classic JVM are interrupted by the garbage collector thread

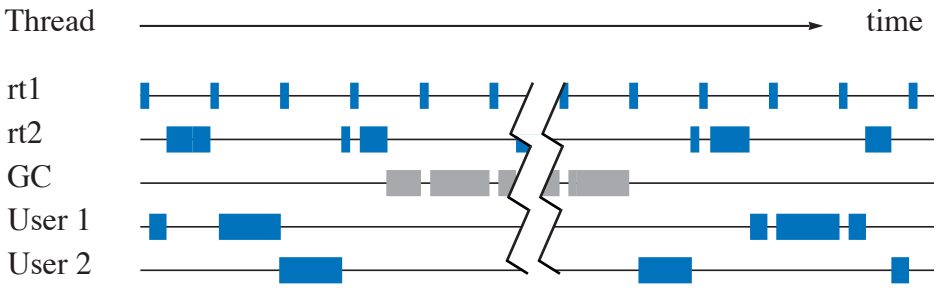


Figure 10.2: RealtimeThreads can interrupt garbage collector activity

`NoHeapRealtimeThread` were defined. These thread types are unaffected, or at least less severely affected, by garbage collection activity. Also, at least 28 new priority levels, logically higher than the priority of the garbage collector, are available for these threads, as illustrated in Fig. 10.2.

10.3.1.2 Memory Management

For realtime threads not to be affected by garbage collector activity, these threads need to use memory areas that are not under the control of the garbage collector. New memory classes, `ImmortalMemory` and `ScopedMemory`, provide these memory areas. One important consequence of using special memory areas is, of course, that the advantages of dynamic memory management is not fully available to realtime threads.

10.3.1.3 Synchronization

In realtime systems with threads of different priority levels, priority inversion situations must be avoided. Priority inversion occurs when a thread of high priority is blocked by waiting for a monitor that is owned by a thread of a lower priority that

is preempted by some thread with intermediate priority. The RTSJ provides two alternatives, priority inheritance and the priority ceiling protocol, to avoid priority inversion.

10.3.1.4 Limitations of the RTSJ and their solution

The RTSJ provides a solution for realtime programming, but it also brings new difficulties to the developer. The most important consequence is that applications have to be split strictly into two parts: a realtime and a non realtime part. Communication between these parts is heavily restricted: realtime threads cannot perform memory operations such as the allocation of objects on the normal heap which is under the control of the garbage collector. Synchronization between realtime and non realtime threads is also severely restricted to prevent realtime threads from being blocked by the garbage collector due to priority inversion.

The JamaicaVM removes these restrictions by using its realtime garbage collection technology. Realtime garbage collection obviates the need to make a strict separation of realtime and non realtime code. Combined with static memory deallocation that automatically replaces some dynamic memory allocations, dynamic allocations in realtime code may even be eliminated completely. Using RTSJ with realtime garbage collection provides necessary realtime facilities without the cumbersomeness of having to segregate a realtime application.

10.3.2 Java Native Interface

Both the need to use legacy code and the desire to access exotic hardware may make it advantageous to call foreign code out of a JVM. The Java Native Interface (JNI) provides this access. JNI can be used to embed code written in other languages than Java, (usually C), into Java programs.

While calling foreign code through JNI is flexible, the resulting code has several disadvantages. It is usually harder to port to other operating systems or hardware architectures than Java code. Another drawback is that JNI is not very high-performing on any Java Virtual Machine. The main reason for the inefficiency is that the JNI specification is independent of the Java Virtual Machine. Significant additional bookkeeping is required to insure that Java references that are handed over to the native code will remain protected from being recycled by the garbage collector while they are in use by the native code. The result is that calling JNI methods is usually expensive.

An additional disadvantage of the use of native code is that the application of any sort of formal program verification of this code becomes virtually intractable.

Nevertheless, because of its availability for many JVMs, JNI is the most popular Java interface for accessing hardware. It can be used whenever Java programs

need to embed C routines that are not called too often or are not overly time-critical. If portability to other JVMs is a major issue, there is no current alternative to JNI. When portability to other operating systems or hardware architectures is more important, RTDA or RTSJ is a better choice for device access.

10.3.3 Java 2 Micro Edition

Usually when one refers to Java, one thinks of the Java 2 Standard Edition (J2SE), but this is not the only Java configuration available. For enterprise applications, Sun Microsystems has defined a more powerful version of Java, Java 2 Enterprise Edition (J2EE), which supports Web servers and large applications. There is also a stripped down version of Java for embedded applications, Java 2 Micro Edition (J2ME). This is interesting for timely Java development on systems with limited resources.

J2ME is in fact not a single Java implementation, but a family of implementations. At its base, J2ME has two configurations: Connected Device Configuration (CDC) and Connected Limited Device Configuration (CLDC). Profiles for particular application domains are layered on top of these configurations, e.g. Mobile Information Device Profile (MIDP) on CLDC, and Personal Profile on CDC.

The JamaicaVM supports both base configurations. Smaller Java configurations are interesting not only on systems with hardware limitations, but also for certification. The vast number of classes in J2SE would make any JVM certification daunting. Again, the choice between J2SE and J2ME is a trade off between flexibility on the one hand and leanness and robustness on the other. A safety-critical version of JamaicaVM may well support a safety-critical profile for CDC in the future.

10.4 Memory Management

In a system that supports realtime garbage collection, RTSJ's strict separation into realtime and non realtime threads is not necessary. The strict splitting of an application is consequently not required. Threads are activated only depending on their priorities, as depicted in Fig. 10.3.

The realtime garbage collector performs its work predictably within the application threads. It is activated when memory is allocated. The work done on an allocation must be preemptible, so that more urgent threads can become active.

The implementation of a realtime garbage collector must solve a number of technical challenges. Garbage collector activity must be performed in very small single increments of work. In the JamaicaVM, one increment consists of processing and possibly reclaiming only 32 bytes of memory. On every allocation, the

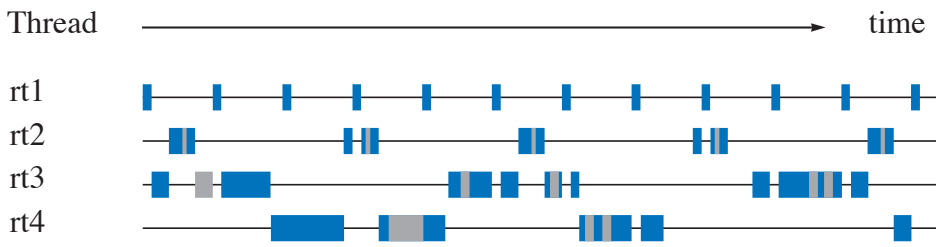


Figure 10.3: JamaicaVM provides realtime behavior for all threads.

allocating thread "pays" for the memory by performing a small number of these increments. The number of increments can be analyzed to determine worst-case behavior for realtime code.

10.4.1 Memory Management of RTSJ

The RTSJ provides a powerful extension to the Java specification. Its full power, however, is achieved only by the combination with a realtime garbage collector that helps to overcome its restrictions. Since JamaicaVM uses a realtime garbage collector, it does not need to impose the limitation that the Real-Time Specification for Java puts onto realtime programming onto realtime applications developed with the JamaicaVM. The limitations that are relaxed in JamaicaVM affect the use of memory areas, thread priorities, runtime checks, and static initializers.

10.4.1.1 Use of Memory Areas

Since Jamaica's realtime garbage collector does not interrupt application threads, `RealtimeThreads` and even `NoHeapRealtimeThreads` are not required to run in their own memory area outside the control of the garbage collector. Instead, any thread can use and access the normal garbage collected heap.

10.4.1.2 Thread priorities

In Jamaica, `RealtimeThreads`, `NoHeapRealtimeThreads` and normal Java `Thread` objects all share the same priority range. The lowest possible thread priority for all of these threads is defined in package `java.lang`, class `Thread` by field `MIN_PRIORITY`. The highest possible priority is can be obtained by querying `instance().getMaxPriority()`, class `PriorityScheduler`, package `javax.realtime`.

10.4.1.3 Runtime checks for NoHeapRealtimeThread

Since even `NoHeapRealtimeThreads` are immune to interruption by garbage collector activities, JamaicaVM does not restrict these threads from accessing objects allocated on the normal heap. Runtime checks that typically ensure that these threads do not access objects allocated on the heap can be disabled in the JamaicaVM. The result is better overall system performance.

10.4.1.4 Static Initializers

In order to permit the initialization of classes even when their first reference is performed within `ScopedMemory` or `ImmortalMemory` within a `RealtimeThread` or `NoHeapRealtimeThread`, and to permit the access of static fields such as `System.out` from within these threads, static initializers are typically executed within `ImmortalMemory` that is accessible by all threads. However, this prevents these objects from being reclaimed when they are no longer in use. This can result in a serious memory leak when dynamic class loading is used since memory allocated by the static initializers of dynamically loaded classes will never be reclaimed.

Since the RTSJ implementation in the JamaicaVM does not limit access to heap objects within any threads, there is no need to execute static initializers within `ImmortalMemory`. However, objects allocated in static initializers typically must be accessible by all threads. Therefore they cannot be allocated in a scoped memory area when this happens to be the current thread's allocation environment when the static initializer is executed.

The JamaicaVM executes all static initializers within heap memory. Objects allocated by static initializers may be accessed by all threads, and they may be reclaimed by the garbage collector. There is no memory leak if classes are loaded dynamically by a user class loader.

10.4.1.5 Class `PhysicalMemoryManager`

Names and instances of class `javax.realtime.PhysicalMemoryTypeFilter` that are passed to method `registerFilter` of the class `javax.realtime.PhysicalMemoryManager` are, by the RTSJ, required to be allocated in immortal memory. Realtime garbage collection obviates this requirement. The JamaicaVM does not enforce it either.

10.4.2 Finalizers

Care needs to be taken when using Java's finalizers. A finalizer is a method that can be redefined by any Java class to perform actions after the garbage collector

has determined that an object has become unreachable. Improper use of finalizers can cause unpredictable results.

The Java specification does not give any guarantees that an object will ever be recycled by the system and that a finalizer will ever be called. Furthermore, if several unreachable objects have a finalizer, the execution order of these finalizers is undefined. For these reasons, it is generally unwise to use finalizers in Java at all. The developer cannot rely on the finalizer ever being executed. Moreover, during the execution of a finalizer, the developer cannot rely on the availability of any other resources since their finalizers may have been executed already.

In addition to these unpredictabilities, the use of finalizers has an important impact on the memory demand of an application. The garbage collector cannot reclaim the memory of any object that has been found to be unreachable before its finalizer has been executed. Consequently, the memory occupied by such objects remains allocated.

The finalizer methods are executed by a finalizer thread, which the JamaicaVM by default runs at the highest priority available to Java threads. If this finalizer thread does not obtain sufficient execution time, or it is stopped by a finalizer that is blocked, the system may run out of memory. In this case, explicit calls to `Runtime.runFinalization()` may be required by some higher priority task to empty the queue of finalizable objects.

The use of finalizers is more predictable for objects allocated in `ScopedMemory` or `ImmortalMemory`. For `ScopedMemory`, all finalizers will be executed when the last thread exits a scope. This may cause a potentially high overhead for exiting this scope. The finalizers of objects that are allocated in `ImmortalMemory` will never be executed.

As an alternative to finalizers, the consequent use of `finally` clauses in Java code to free unused resources at a predefined time is highly recommended. Using finalizers may be helpful during debugging to find programming bugs like leakage of resources or to visualize when an object's memory is recycled. In a production release, any finalizers (even empty ones) should be removed due to the impact they have on the runtime and the potential for memory leaks caused by their presence.

10.4.3 Configuring a Realtime Garbage Collector

To be able to determine worst-case execution times for memory allocation operations in a realtime garbage collector, one needs to know the memory required by the realtime application. With this information, a worst-case number of garbage collector increments that are required on an allocation can be determined (see § 7). Automatic tools can help to determine this value. The heap size can then be selected to give sufficient headroom for the garbage collector, while a larger heap size ensures a shorter execution time for allocation. Tools like the analyzer in the

JamaicaVM help to configure a system and find suitable heap size and allocation times.

10.4.4 Programming with the RTSJ and Realtime Garbage Collection

Once the unpredictability of the garbage collector has been solved, realtime programming is possible even without the need for special thread classes or the use of specific memory areas for realtime code.

10.4.4.1 Realtime Tasks

In Jamaica, garbage collection activity is performed within application threads and only when memory is allocated by a thread. A direct consequence of this is that any realtime task that performs no dynamic memory allocation will be entirely unaffected by garbage collection activity. These realtime tasks can access objects on the normal heap just like all other tasks. As long as realtime tasks use a priority that is higher than other threads, they will be guaranteed to run when they are ready. Furthermore, even realtime tasks may allocate memory dynamically. Just like any other task, garbage collection work needs to be performed to pay for this allocation. Since a worst-case execution time can be determined for the allocation, the worst-case execution time of the task that performs the allocation can be determined as well.

10.4.4.2 Communication

The communication mechanisms that can be used between threads with different priority levels and timing requirements are basically the same mechanisms as those used for normal Java threads: shared memory and Java monitors.

Shared Memory Since all threads can access the normal, garbage-collected heap without suffering from unpredictable pauses due to garbage collector activity, this normal heap can be used for shared memory communication between all threads. Any high priority task can access objects on the heap even while a lower priority thread accesses the same objects or even while a lower priority thread allocates memory and performs garbage collection work. In the latter case, the small worst-case execution time of an increment of garbage collection work ensures a bounded and small thread preemption time, typically in the order of a few microseconds.

Synchronization The use of Java monitors in `synchronized` methods and explicit `synchronized` statements enables atomic accesses to data structures. These mechanisms can be used equally well to protect accesses that are performed in high priority realtime tasks and normal non-realtime tasks. Unfortunately, the standard Java semantics for monitors does not prevent priority inversion that may result from a high priority task trying to enter a monitor that is held by another task of lower priority. The stricter monitor semantics of the RTSJ avoid this priority inversion. All monitors are required to use priority inheritance or the priority ceiling protocol, such that no priority inversion can occur when a thread tries to enter a monitor. As in any realtime system, the developer has to ensure that the time that a monitor is held by any thread must be bounded when this monitor needs to be entered by a realtime task that requires an upper bound for the time required to obtain this monitor.

10.4.4.3 Standard Data Structures

The strict separation of an application into a realtime and non-realtime part that is required when the Real-Time Specification for Java is used in conjunction with a non-realtime garbage collector makes it very difficult to have global data structures that are shared between several tasks. The Real-Time Specification for Java even provides special data structures such as `WaitFreeWriteQueue` that enable communication between tasks. These queues do not need to synchronize and hence avoid running the risk of introducing priority inversion. In a system that uses realtime garbage collection, such specific structures are not required. High priority tasks can share standard data structures such as `java.util.Vector` with low priority threads.

10.4.5 Memory Management Guidelines

The JamaicaVM provides four options for memory management: `ImmortalMemory`, `ScopedMemory`, static memory deallocation, and realtime dynamic garbage collection on the normal heap. They may all be used freely. The choice of which to use is determined by what the best trade off between external requirements, compatibility, and efficiency for a given application.

`ImmortalMemory` is in fact quite dangerous. Memory leaks can result from improper use. Its use should be avoided unless compatibility with other RTSJ JVMs is paramount or heap memory is not allowed by the certification regime required for the project.

`ScopedMemory` is safer, but it is generally inefficient due to the runtime checks required by its use. When a memory check fails, the result is a runtime exception, which is also undesirable in safety-critical code. In many cases, static

memory de-allocation can do the same job without the runtime checks. Escape analysis ensures that the checks are not needed and that no memory related exception is ever thrown by the corresponding memory access. Therefore, static memory deallocation is generally a better solution than `ScopedMemory`.

When static memory deallocation is not applicable, one can always fall back on the realtime garbage collector. It is both safe and relatively efficient. Still any heap allocation has an associated garbage collection time penalty. Realtime garbage collection makes an allocating thread pay the penalty up front.

One important property of the JamaicaVM is that any realtime code that runs at high priority and that does not perform memory allocation is guaranteed not to be delayed by garbage collection work. This important feature holds for standard RTSJ applications only under the heavy restrictions that apply to `NoHeapRealtimeThreads`.

10.5 Scheduling and Synchronization

As the reader may have already noticed in the previous sections, scheduling and synchronization are closely related. Scheduling threads that do not interact is quite simple; however, interaction is necessary for sharing data among cooperating tasks. This interaction requires synchronization to ensure data integrity. There are implications on scheduling of threads and synchronization beyond memory access issues.

10.5.1 Schedulable Entities

The RTSJ introduces new scheduling entities to Java. `RealtimeThread` and `NoHeapRealtimeThread` are thread types with clearer semantics than normal Java threads of class `Thread` and additional scheduling possibilities. Events are the other new thread-like construct used for transient computations. To save resources (mainly operating system threads, and thus memory and performance), `AsyncEvents` can be used for short code sequences instead. They are easy to use because they can easily be triggered programmatically, but they must not be used for blocking. Also, there are `BoundAsyncEvents` which each require their own thread and thus can be used for blocking. They are as easy to use as normal `AsyncEvents`, but do not use fewer resources than normal threads. `AsyncEventHandlers` are triggered by an asynchronous event. All three execution environments, `RealtimeThreads`, `NoHeapRealtimeThreads` and `AsyncEventHandlers`, are schedulable entities, i.e., they all have release parameters and scheduling parameters that are considered by the scheduler.

10.5.1.1 RealtimeThreads and NoHeapRealtimeThreads

The RTSJ includes new thread classes `RealtimeThreads` and `NoHeapRealtimeThreads` to improve the semantics of threads for realtime systems. These threads can use a priority range that is higher than that of all normal Java `Threads` with at least 28 unique priority levels. The default scheduler uses these priorities for fixed priority, preemptive scheduling. In addition to this, the new thread classes can use the new memory areas `ScopedMemory` and `ImmortalMemory` that are not under the control of the garbage collector.

As previously mentioned, threads of class `NoHeapRealtimeThreads` are not permitted to access any object that was allocated on the garbage collected heap. Consequently, these threads do not suffer from garbage collector activity as long as they run at a priority that is higher than that of any other schedulable object that accesses the garbage collected heap. In the JamaicaVM Java environment, the memory access restrictions present in `NoHeapRealtimeThreads` are not required to achieve realtime guarantees. Consequently, the use of `NoHeapRealtimeThreads` is neither required nor recommended.

Apart from the extended priority range, `RealtimeThreads` provide features that are required in many realtime applications. Scheduling parameters for periodic tasks, deadlines, and resource constraints can be given for `RealtimeThreads`, and used to implement more complex scheduling algorithms. For instance, periodic threads in the JamaicaVM use these parameters. In the JamaicaVM Java environment, normal Java threads also profit from strict fixed priority, preemptive scheduling; but for realtime code, the use of `RealtimeThread` is still recommended.

10.5.1.2 AsyncEventHandlers vs. BoundAsyncEventHandlers

An alternative execution environment is provided through classes `AsyncEventHandler` and `BoundAsyncEventHandler`. Code in an event handler is executed to react to an event. Events are bound to some external happening (e.g, a processor interrupt), which triggers the event.

`AsyncEventHandler` and `BoundAsyncEventHandler` are schedulable entities that are equipped with release and scheduling parameters exactly as `RealtimeThread` and `NoHeapRealtimeThread`. The priority scheduler schedules both threads and event handlers, according to their priority. Also, admission checking may take the release parameters of threads and asynchronous event handlers in account. The release parameters include values such as execution time, period, and minimum interarrival time.

One important difference from threads is that an `AsyncEventHandler` is not bound to one single thread. This means, that several invocations of the same

handler may be performed in different thread environments. A pool of preallocated `RealtimeThreads` is used for the execution of these handlers. Event handlers that may execute for a long time or that may block during their execution may block a thread from this pool for a long time. This may make the timely execution of other event handlers impossible.

Any event handler that may block should therefore have one `RealtimeThread` that is assigned to it alone for the execution of its event handler. Handlers for class `BoundAsyncEventHandler` provide this feature. They do not share their thread with any other event handler and they may consequently block without disturbing the execution of other event handlers.

Due to the additional resources required for a `BoundAsyncEventHandler`, their use should be restricted to blocking or long running events only. The sharing of threads used for normal `AsyncEventHandlers` permits the use of a large number of event handlers with minimal resource usage.

10.5.2 Synchronization

Synchronization is essential to data sharing, especially between cooperating realtime tasks. Passing data between threads at different priorities without impairing the realtime behavior of the system is the most important concern. It is essential to ensure that a lower priority task cannot preempt a higher priority task.

The situation in Fig. 10.4 depicts a case of priority inversion when using monitors, the most common priority problem. The software problems during the Pathfinder mission on Mars is the most popular example of a classic priority inversion error (see Michael Jones' web page [5]).

In this situation, a higher priority thread A has to wait for a lower priority thread B because another thread C with even lower priority is holding a monitor for which A is waiting. In this situation, B will prevent A and C from running, because A is blocked and C has lower priority. In fact, this is a programming error. If a thread might enter a monitor which a higher priority thread might require, then no other thread should have a priority in between the two.

Since errors of this nature are very hard to locate, the programming environment should provide a means for avoiding priority inversion. The RTSJ defines two possible mechanisms for avoiding priority inversion: Priority Inheritance and Priority Ceiling Emulation. The JamaicaVM implements both mechanisms.

10.5.2.1 Priority Inheritance

Priority Inheritance is a protocol which is easy to understand and to use, but that poses the risk of causing deadlocks. If priority inheritance is used, whenever a higher priority thread waits for a monitor that is held by a lower priority thread,

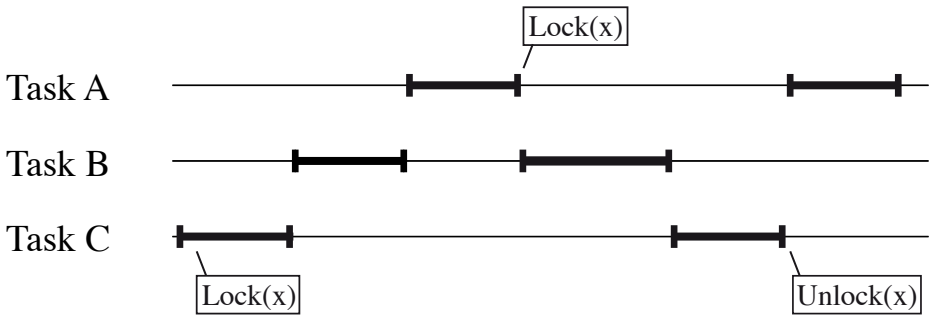


Figure 10.4: Priority Inversion

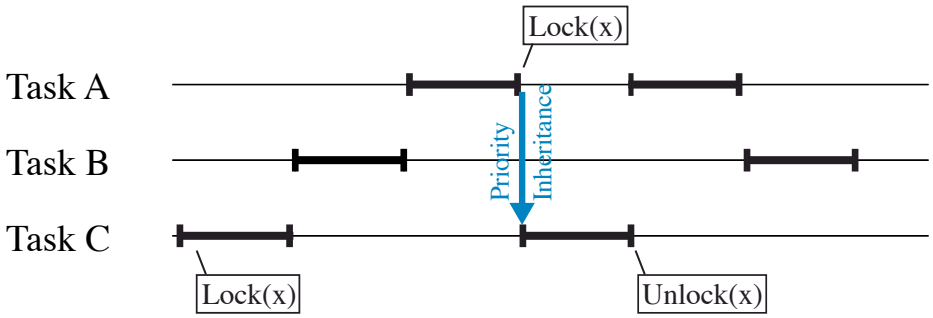


Figure 10.5: Priority Inheritance

the lower priority thread's priority is boosted to the priority of the blocking thread. Fig. 10.5 illustrates this.

10.5.2.2 Priority Ceiling Emulation

Priority Ceiling Emulation is widely used in safety-critical system. The priority of any thread entering a monitor is raised to the highest priority of any thread which could ever enter the monitor. Fig. 10.6 illustrates the Priority Ceiling Emulation protocol.

As long as no thread that holds a priority ceiling emulation monitor blocks, any thread that tries to enter such a monitor can be sure not to block.[1] Consequently, the use of priority ceiling emulation automatically ensures that a system is deadlock-free.

[1] If any other thread owns the monitor, its priority will have been boosted to the ceiling priority. Consequently, the current thread cannot run and try to enter this monitor.

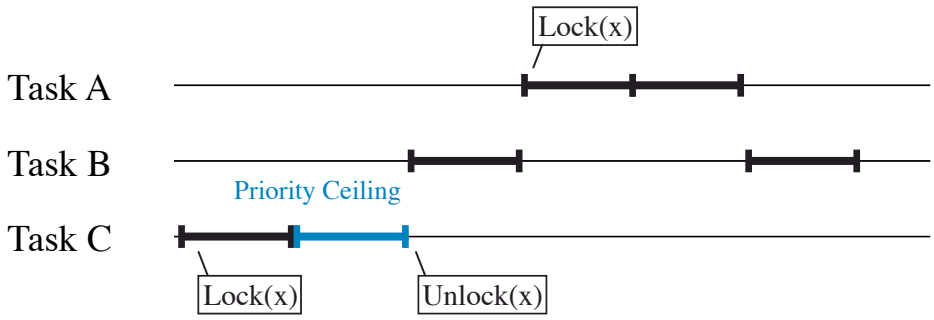


Figure 10.6: Priority Ceiling Emulation Protocol

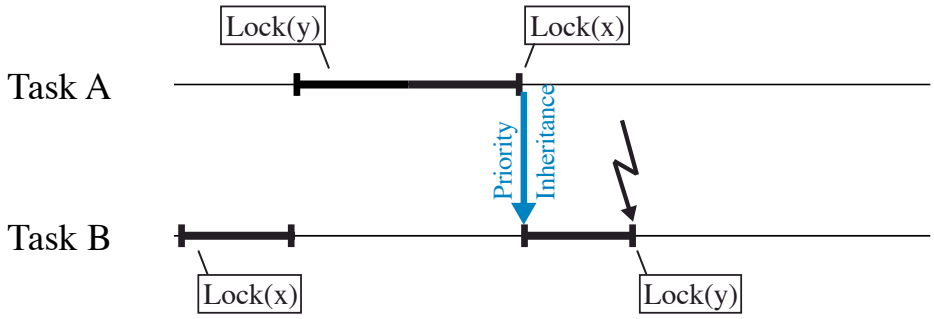


Figure 10.7: Deadlocks are possible with Priority Inheritance

10.5.2.3 Priority Inheritance vs. Priority Ceiling Emulation

Priority Inheritance should be used with care, because it can cause deadlocks when two threads try to enter the same two monitors in different order. This is shown in Fig. 10.7. Thus it is safer to use Priority Ceiling Emulation, since when used correctly, deadlocks cannot occur there. Priority Inheritance deadlocks can be avoided, if all programmers make sure to always enter monitors in the same order.

Unlike classic priority ceiling emulation, the RTSJ permits blocking while holding a priority ceiling emulation monitor. Other threads that may want to enter the same monitor will be stopped exactly as they would be for a normal monitor. This fall back to standard monitor behavior permits the use of priority ceiling emulation even for monitors that are used by legacy code.

The advantage of a limited and short execution time for entering a priority ceiling monitor, working on a shared resource, then leaving this monitor are, however, lost when a thread that has entered this monitor may block. Therefore the system designer should restrict the use of priority ceiling monitors to short code sequences that only access a shared resource and that do not block. Entering and exiting the

monitor can then be performed in constant time, and the system ensures that no thread may try to enter a priority ceiling monitor that is held by some other thread.

Since priority ceiling emulation requires adjusting a thread's priority every time a monitor is entered or exited, there is an additional runtime overhead for this priority change when using this kind of monitors. This overhead can be significant compared to the low runtime overhead that is incurred to enter or leave a normal, priority inheritance monitor. In this case, there is a priority change penalty only when a monitor has already been taken by another thread.

Future versions of the Jamaica Java implementation may optimize priority ceiling and avoid unnecessary priority changes. The JamaicaVM uses atomic code sequences and restricts thread switches to certain points in the code. A synchronized code sequence that is protected by a priority ceiling monitor and that does not contain a synchronization point may not require entering and leaving of the monitor at all since the code sequence is guaranteed to be executed atomically due to the fact that it does not contain a synchronization point.

10.5.3 Scheduling Policy and Priorities

Although JamaicaVM uses its own scheduler, the realtime behavior depends heavily on the scheduling policy of the underlying operating system. Best results can be achieved by using priority based scheduling using a *first-in-first-out* scheduling policy since this corresponds to the scheduling policy implemented by JamaicaVM's own scheduler.

10.5.3.1 Native Priorities

In JamaicaVM, a priority map defines which native (OS) priorities are used for the different Java thread priorities. This priority map can be set via the environment variable `JAMAICAVM_PRIMAP` (see § 13.3), or using the Jamaica Builder via the `-priMap` option (see § 14).

Normal (non-realtime) Java thread priorities should usually be mapped to a single OS priority since otherwise lower priority Java threads may receive no CPU time if a higher priority thread is running constantly. The reason for this is that legacy Java code that expects lower priority threads to run even if higher priorty threads are ready may not work otherwise. A *fairness* mechanism in JamaicaVM is used only for the lowest Java thread priorities that map to the same OS priority. For applications written to work with *first-in-first-out* scheduling, mapping different Java priorities to different OS priorities, however, can result in better performance.

Higher Java priorities used for instances of `RealtimeThread` and `AsyncEventHandler`, usually the Java priorities 11 through 38, should be mapped to

distinct priorities of the underlying OS. If there are not sufficiently many OS priority levels available, different Java priorities may be mapped to the same native priority. The Jamaica scheduler will still run the thread with higher Java priority before running the lower priority threads. However, having the same native priority may result in higher thread-switch overhead since the underlying OS does not know about the difference in Java priorities and may attempt to run the wrong thread.

The highest Java priority, usually 39, is used for the synchronization thread and is not available to Java code. This thread manages time slicing between the normal Java threads, so this should usually be mapped to a value that is higher or equal to the native priority used for Java priority 10, the maximum priority for normal, non-realtime Java threads. Using a higher priority for the synchronization thread may introduce jitter to the realtime threads, while using a lower value will disable time slicing and fairness for this and higher priorities.

10.5.3.2 POSIX Scheduling Policies

On POSIX systems, the scheduling policy can be set via the environment variable `JAMAICAVM_SCHEDULING_POLICY` (see § 13.3). Using the Jamaica Builder, the scheduling policy can be set with the `-schedulingPolicy` option (see § 14). These are the supported POSIX scheduling policies:

- `OTHER` — default scheduling
- `FIFO` — first in first out
- `RR` — round robin

The default is `OTHER`, which may not be a realtime policy depending on the target OS. To obtain realtime performance, the use of `FIFO` is required. Using `RR` is an alternative to `FIFO`, but it does make sense only in case Jamaica threads are supposed to share the CPU with other processes running at the same priority. Using `FIFO` or `RR` requires superuser privileges (root access) on some systems, e.g., Linux.

Scheduling policies `FIFO` and `RR` require native thread priorities that are `1` or larger, while the default priority map used by JamaicaVM may map all Java thread priorities to native priority `0` if this is a legal priority for the `OTHER` policy (e.g., on Linux). Hence, it is required to define a different priority map if these scheduling policies are used.

Native priorities that are lower than the minimum priority of the selected scheduling policy (e.g., priority `0` is lower than the minimum `FIFO` priority which is `1`) are implemented by falling back to the `OTHER` scheduling policy for the affected threads.

On Linux, `FIFO` scheduling is recommended for `RealtimeThread`s and `AsyncEventHandler`s and `OTHER` for normal Java threads. These are the corresponding settings:

```
JAMAICAVM_SCHEDULING_POLICY=FIFO
JAMAICAVM_PRIMAP=1..10=0,11..38=1..28,39=0
```

The result is that a Java application that uses only normal Java threads will use `OTHER` scheduling and run in user mode, while any threads and event handlers that use RTSJ's realtime priorities (11 through 38) will use the corresponding `FIFO` priorities. The highest Java priority, 39, is used for the synchronization thread. This thread manages time slicing between the normal Java threads, so this can use the `OTHER` scheduling policy as well.

10.6 Libraries

The use of a standard Java libraries within realtime code poses severe difficulties, since standard libraries typically are not developed with the strict requirements on execution time predictability that come with the use in realtime code. For use within realtime applications, any libraries that are not specifically written and documented for realtime system use cannot be used without inspection of the library code.

The availability of source code for standard libraries is an important prerequisite for their use in realtime system development. Within the JamaicaVM, large parts of the standard Java APIs are taken from OpenJDK, which is an open source project. The source code is freely available, so that the applicability of certain methods within realtime code can be checked easily.

10.7 Summary

As one might expect, programming realtime systems in Java is more complicated than standard Java programming. A realtime Java developer must take care with many Java constructs. With timely Java development using JamaicaVM, there are instances where a developer has more than one possible implementation construct to choose from. Here, the most important of these points are recapitulated.

10.7.1 Efficiency

All method calls and interface calls are performed in constant time. They are almost as efficient as C function calls, so do not avoid them except in places where one would avoid a C function call as well.

When accessing final `local` variables or private fields from within inner classes in a loop, one should generally cache the result in a local variable for performance reasons. The access is in constant time, but slower than normal local variables.

Using the String operator + causes memory allocation with an execution time that is linear with regard to the size of the resulting String. Using array initialization causes dynamic allocations as well.

For realtime critical applications, avoid static initializers or explicitly call the static initializer at startup. When using a java compiler earlier than version 1.5, the use of *classname* `.class` causes dynamic class loading. In realtime applications, this should be avoided or called only during application startup. Subsequent usage of the same class will then be cached by the JVM.

10.7.2 Memory Allocation

The RTSJ introduces new memory areas such as `ImmortalMemoryArea` and `ScopedMemory`, which are inconvenient for the programmer, and at the same time make it possible to write realtime applications that can be executed even on virtual machines without realtime garbage collection.

In JamaicaVM, it is safe, reliable, and convenient to just ignore those restrictions and rely on the realtime garbage collection instead. Be aware that if extensions of the RTSJ without sticking to restrictions imposed by the RTSJ, the code will not run unmodified on other JVMs. To make sure code is portable, one should use the `-strictRTSJ` switch. StrictRTSJ mode can safely be used by any Java program without modifications.

10.7.3 EventHandlers

`AsyncEventHandlers` should be used for tasks that are triggered by some external event. Many event handlers can be used simultaneously; however, they should not block or run for a long time. Otherwise the execution of other event handlers may be blocked.

For longer code sequences, or code that might block, event handlers of class `BoundAsyncEventHandler` provide an alternative that does not prevent the execution of other handlers at the cost of an additional thread.

The scheduling and release parameters of event handlers should be set according to the scheduling needs for the handler. Particularly, when rate monotonic analysis [8] is used, an event handler with a certain minimal interarrival time should be assigned a priority relative to any other events or (periodic) threads using this minimal interarrival time as the period of this schedulable entity.

10.7.4 Monitors

Priority Inheritance is the default protocol in the RTSJ. It is safe and easy to use, but one should take care to nest monitor requests properly and in the same order in all threads. Otherwise, it can cause deadlocks. When used properly, Priority Ceiling Emulation (PCE) can never cause deadlocks, but care has to be taken that a monitor is never used in a thread of higher priority than the monitor. Both protocols are efficiently implemented in the JamaicaVM.

Chapter 11

Multicore Guidelines

While on single-core systems multithreaded computation eventually boils down to the sequential execution of instructions on a single CPU, multicore systems pose new challenges to programmers. This is especially true for languages that expose features of the target hardware relatively directly, such as C. For example, shared memory communication requires judiciously placed memory fences to prevent compiler optimizations that can lead to values being created "out of thin air".

High-level languages such as Java, which has a well-defined and machine-independent memory model [4, Chapter 17], shield programmers from such surprises. In addtion, high-level languages provide automatic memory management. The Jamaica multicore VM provides concurrent, parallel, real-time garbage collection:

Concurrent Garbage collection can take place on some CPUs while other CPUs execute application code.

Parallel Several CPUs can perform garbage collection at the same time.

Real-time There is a guaranteed upper bound on the amount of time any part of application code may be suspended for garbage collection work. At the same time, it is guaranteed that garbage collection work will be sufficient to reclaim enough memory so all allocation requests by the application can be satisfied.

JamaicaVM's garbage collector achieves hard real-time guarantees by carefully distributing the garbage collection to all available CPUs [9].

11.1 Tool Usage

For versions of JamaicaVM with multicore support the Builder can build applications with and without multicore support. This is controlled via the Builder option

`-parallel`. On systems with only one CPU or for applications that cannot benefit from parallel execution, multicore support should disabled. The multicore version has a higher overhead of heap memory than the single-core version (see § C).

In order to limit the CPUs used by Jamaica, a set of CPU affinities may be given to the Builder or VM via the option `-Xcpus`. See § 13.1.2 and § 14.3 for details. While Jamaica supports all possible subsets of the existing CPUs, operating systems may not support these. The set of all CPUs and all singleton sets of CPUs are usually supported, though. For more information, please consult the documentation of the operating system you use.

To find out whether a particular Jamaica virtual machine provides multicore support, use the `-version` option. A VM with multicore support will identify itself as `parallel`.

11.2 Setting Thread Affinities

On a multicore system, by default the scheduler can assign any thread to any CPU as long as priorities are respected. In many cases this flexibility leads to reduced throughput or increased jitter. The main reason is that migrating a thread form one CPU to another is expensive: it renders the code and data stored in the cache useless, which delays execution. Reducing the scheduler's choice by "pinning" a thread to a specific CPU can help. In JamaicaVM the RTSJ class `javax.realtime.AffinitySet` enables programmers to restrict on which CPUs a thread can run. The following sections present rules of thumb for choosing thread affinities in common situations. In practice, usually experimentation is required to see which affinities work best for a particular application.

11.2.1 Communication through Shared Memory

Communication of threads through shared memory is usually more efficient if both threads run on the same CPU. This is because threads on the same CPU can communicate via the CPU's cache, while in order for data to pass from one CPU to another, it has to go via main memory, which is slower. The decision on whether pinning two communicating threads to the same or to different CPUs should be based on the tradeoff between computation and communication: if computation dominates, it will usually be better to use different CPUs; if communication dominates, using the same CPU will be better.

Interestingly, the same effect can also occur for threads that do not communicate, but that write data in the same cache line. This is known as *false sharing*.

In JamaicaVM this can occur if two threads modify data in the same object (more precisely, the same block).

11.2.2 Performance Degradation on Locking

If two contenders for the same monitor can only run on the same CPU, the runtime system may be able to decide more efficiently whether the monitor is free and may be acquired (i.e., *locked*). Consider the following scenario:

- A high-priority thread A repeatedly acquires and releases a monitor.
- A low-priority thread B repeatedly acquires and releases the same monitor.

This happens, for example, if A and B concurrently read fields of a synchronized data-structure.

Assume that thread B is started and later also thread A. At some point, A may have to wait until B releases the monitor. Then A resumes. Since A is of higher priority than B, A will not be preempted by B. If A and B are tied to the same CPU this means that B cannot run while A is running. If A releases the monitor and tries to re-acquire it later, it is clear that it cannot have been taken by B in the meantime. Since the monitor is free, it can be taken immediately, which is very efficient.

If, on the other hand, A and B can run on different CPUs, B can be running while A is running, and it may acquire the monitor when A releases it. In this case, A has to re-obtain the monitor from B before it can continue. The additional overhead for blocking A and for waking up A after B has released the monitor can be significant.

11.2.3 Periodic Threads

Some applications have periodic events that need to happen with high accuracy. If this is the case, cache latencies can get into the way. Consider the following scenario:

- A high-priority thread A runs every 2ms for 1ms and
- A low-priority thread B runs every 10ms for 2ms.

If both threads run on the same CPU, B will fill some of the gaps left by A. For the gaps filled by B, when A resumes, it first needs to fill the cache with its own code and data. This can lead to *CPU stalls*. These stalls only occur when B did run immediately before A. They do not occur after the gaps during which the CPU was idle. The fact that stalls occur sometimes but sometimes not will be observed

as jitter in thread A. The problem can be alleviated by tying A and B to different CPUs.

11.2.4 Rate-Monotonic Analysis

Rate-monotonic analysis is a technique for determining whether a scheduling problem is feasible on a system with thread preemption such that deterministic response times can be guaranteed with simple (rate-monotonic) scheduling algorithms. Rate-monotonic analysis only works for single-core systems. However, if a subset of application threads can be identified that have little depencency on the other application threads it may be possible to schedule these based on rate-monotonic analysis.

A possible scenario where this can be a useful approach is an application where some threads guarantee deterministic responses of the system, while other threads perform data processing in the background. The subset of threads in charge of deterministic responses could be isolated to one CPU and rate-monotonic scheduling could be used for them.

11.2.5 The Operating System's Interrupt Handler

Operating systems usually tie interrupt handling to one particular CPU. Cache effects described in § 11.2.3 above can also occur between the interrupt handling code and application threads. Therefore, jitter may be reduced by running application threads on CPUs other than the one in charge of the operating system's interrupt handling.

Part III

Tools Reference

Chapter 12

The Jamaica Java Compiler

The command `jamaicac` is a compiler for the Java programming language and is based on OpenJDK's Java Compiler. It uses the system classes of the Jamaica distribution as default bootclasspath.

12.1 Usage of jamaicac

The command line syntax for the `jamaicac` is as follows:

`jamaicac [`*options*`] [`*source files and directories*`]`

If directories are specified their source contents are compiled. The command line options of `jamaicac` are those of `javac`. As notable difference, the additional `useTarget` option enables specifying a particular target platform.

12.1.1 Classpath options

Option `-useTarget` *platform*

The `useTarget` option specifies the target platform to compile for. It is used to compute the bootclasspath in case `bootclasspath` is omitted. By default, the host platform is used.

Option `-cp (-classpath)` *path*

The `classpath` option specifies the location for application classes and sources. The path is a list of directories, zip files or jar files separated by the platform specific separator (usually colon, ':'). Each directory or file can specify access rules for types between '[' and ']' (e.g. "`[-X.java]`" to deny access to type `X`).

Option `-bootclasspath` *path*

This option is similar to the option `classpath`, but specifies locations for system classes.

Option `-sourcepath` *path*

The `sourcepath` option specifies locations for application sources. The path is a list of directories. For further details, see option `classpath` above.

Option `-extdirs` *dirs*

The `extdirs` option specifies location for extension zip/jar files, where *path* is a list of directories.

Option `-d` *directory*

The `d` option sets the destination directory to write the generated class files to. If omitted, no directory is created.

12.1.2 Compliance options

Option `-source` *version*

Provide source compatibility for specified version, e.g. 1.6 (or 6 or 6.0).

Option `-target` *version*

Generated class files for a specific VM version, e.g. 1.6 (or 6 or 6.0).

12.1.3 Warning options

Option `-deprecation`

The `deprecation` option checks for deprecation outside deprecated code.

Option `-nowarn`

The `nowarn` option disables all warnings.

12.1.4 Debug options

Option `-g`

The `g` option without parameter activates all debug info.

Option `-g:none`

The `g` option with `none` disables debug info.

Option `-g:{lines,vars,source}`

The `g` option is used to customize debug info.

12.1.5 Other options

Option `-encoding` *encoding*

The `encoding` option specifies custom encoding for all sources. May be overridden for each file or directory by suffixing with '['*encoding*']' (e.g. "X.java[utf8]").

Option `-J`*option*

This option is ignored.

Option `-X`

The `X` option prints non-standard options and exits.

12.2 Environment Variables

The following environment variables control `jamaicac`.

`JAMAICAC_HEAPSIZE` Initial heap size of the `jamaicac` command itself in bytes. Setting this to a larger value will improve the `jamaicac` performance.

`JAMAICAC_MAXHEAPSIZE` Maximum heap size of the `jamaicac` command itself in bytes. If the initial heap size is not sufficient, it will increase its heap dynamically up to this value. To compile large applications, you may have to set this maximum heap size to a larger value.

`JAMAICAC_JAVA_STACKSIZE` Java stack size of the `jamaicac` command itself in bytes.

JAMAICAC_NATIVE_STACKSIZE Native stack size of the `jamaicac` command itself in bytes.

Chapter 13

The Jamaica Virtual Machine Commands

The Jamaica virtual machine provides a set of commands that permit the execution of Java applications by loading a set of class files and executing the code. The command `jamaicavm` launches the standard Jamaica virtual machine. Its variants `jamaicavm_slim`, `jamaicavmp` and `jamaicavmdi` provide special features like debug support.

13.1 jamaicavm — the Standard Virtual Machine

The `jamaicavm` is the standard command to execute non-optimized Java applications in interpreted mode. Its input syntax follows the conventions of Java virtual machines.

`jamaicavm [`*options*`] `*class*` [`*args...*`]`
`jamaicavm [`*options*`] -jar `*jarfile*` [`*args...*`]`

The program's main class is either given directly on the command line, or obtained from the manifest of a Java archive file if option `-jar` is present.

The main class must be given as a qualified class name that includes the complete package path. For example, if the main class `MyClass` is in package `com.mycompany`, the fully qualified class name is `com.mycompany.MyClass`. In Java, the package structure is reflected by nested folders in the file system. The class file `MyClass.class`, which contains the main class's byte code, is expected in the folder `com/mycompany` (or `com\mycompany` on Windows systems). The command line for this example is

```
jamaicavm com.mycompany.MyClass
```

on Unix and Windows systems alike.

The available command line options of `jamaicavm`, are explained in the following sections. In addition to command line options, there are environment variables and Java properties that control the VM. For details on the environment variables, see § 13.3, for the Java properties, see § 13.4.

13.1.1 Command Line Options

Option `-classpath (-cp)` *path*

The `classpath` option sets the search path for class files. The argument must be a list of directories or JAR/ZIP files separated by the platform dependent path separator char (':' on Unix-Systems, ';' on Windows). If this option is not used, the search path for class files defaults to the current working directory.

Option `-D`*name*`=`*value*

The `D` option sets a system property with a given name to a given value. The value of this property will be available to the Java application via functions such as `System.getProperty()`.

Option `-version`

The `version` option prints the version of JamaicaVM.

Option `-help (-?)`

The `help` option prints a short help summary on the usage of JamaicaVM and lists the default values is uses. These default values are target specific. The default values may be overridden by command line options or environment variable settings. Where command line options (set through `-X`*option*) and environment variables are possible, the command line settings have precedence. For the available command line options, see § 13.1.2 or invoke the VM with `-xhelp`.

Option `-xhelp (-X)`

The `xhelp` option prints a short help summary on the extended options of JamaicaVM.

13.1.2 Extended Command Line Options

JamaicaVM supports a number of extended options. Some of them are supported for compatibility with other virtual machines, while some provide functionality that is only available in Jamaica . Please note that the extended options may change without notice. Use them with care.

Option `-Xbootclasspath:`*path*

The `Xbootclasspath` option sets bootstrap search paths for class files. The argument must be a list of directories or JAR/ZIP files separated by the platform dependent path separator char (':' on Unix-Systems, ';' on Windows). Note that the `jamaicavm` command has all boot and standard API classes built in. The boot-classpath has the built-in classes as an implicit first entry in the path list, so it is not possible to replace the built-in boot classes by other classes which are not built-in. However, the boot class path may still be set to add additional boot classes. For commands `jamaicavm_slim`, `jamaicavmp`, etc. that do not have any built-in classes, setting the boot-classpath will force loading of the system classes from the directories provided in this path. However, extreme care is required: The virtual machine relies on some internal features in the boot-classes. Thus it is in general not possible to replace the boot classes by those of a different virtual machine or even by those of another version of the Jamaica virtual machine or even by those of a different Java virtual machine. In most cases, it is better to use `-Xbootclasspath/a`, which appends to the bootstrap class path.

Option `-Xbootclasspath/a:`*path*

The `Xbootclasspath/a` option appends to the bootstrap class path. The argument must be a list of directories or JAR/ZIP files separated by the platform dependent path separator char (':' on Unix Systems, ';' on Windows). For further information, see the `Xbootclasspath` option above.

Option `-Xbootclasspath/p:`*path*

The `Xbootclasspath/p` option prepends to the bootstrap class path. The argument must be a list of directories or JAR/ZIP files separated by the platform dependent path separator char (':' on Unix Systems, ';' on Windows). For further information, see the `Xbootclasspath` option above.

Option `-Xcpus`*cpus*

Specifies the set of CPUs to use, e.g., `0,1,3` will use the CPUs with ids 0, 1, and 3. `-Xcpusall` will use all available CPUs. This option is only available on configurations with multicore support. Be aware that multicore support requires an extra license.

Option `-Xms(-ms)`*size*

The `Xms` option sets initial Java heap size, the default setting is `2M`. This option takes precedence over a heap size set via an environment variable.

Option `-Xmx(-mx)`*size*

The `Xmx` option sets maximum Java heap size, the default setting is `256M`. This option takes precedence over a maximum heap size set via an environment variable.

Option `-Xmi(-mi)`*size*

The `Xmi` option sets heap size increment, the default setting is `4M`. This option takes precedence over a heap size increment set via an environment variable.

Option `-Xss(-ss)`*size*

The `Xss` option sets stack size (native and interpreter). This option takes precedence over a stack size set via an environment variable.

Option `-Xjs(-js)`*size*

The `Xjs` option sets interpreter stack size, the default setting is `64K`. This option takes precedence over a java stack size set via an environment variable.

Option `-Xns(-ns)`*size*

The `Xns` option sets native stack size, set default setting is `64K`. This option takes precedence over a native stack size set via an environment variable.

Option `-Xprof`

Collect simple profiling information using periodic sampling. This profile is used to provide an estimate of the methods which use the most CPU time during the execution of an application. During each sample, the currently executing method

is determined and its sample count is incremented, independent of whether the method is currently executing or is blocked waiting for some other event. The total number of samples found for each method are printed when the application terminates. Note that compiled methods may be sampled incorrectly since they do not necessarily have a stack frame. We therefore recommend to use `Xprof` only for interpreted applications.

This option should not be confused with the profiling facilities provided by `jamaicavmp` (see § 13.2.3).

Option `-Xcheck:jni`

Enable argument checking in the Java Native Interface (JNI). With this option enabled the JamaicaVM will be halted if a problem is detected. Enabling this option will cause a performance impact for the JNI. Using this option is recommended while developing applications that use native code.

13.2 Variants of jamaicavm

A number of variants of the standard virtual machines are provided for special purposes. Their features and uses are described in the following sections. All variants accept the command line options, properties and environment variables of the standard VM. Some variants accept additional command line options as specified below.

13.2.1 jamaicavm_slim

`jamaicavm_slim` is a variant of the `jamaicavm` command that has no built-in standard library classes. Instead, it has to load all standard library classes that are required by the application from the target-specific `rt.jar` provided in the JamaicaVM installation.

Compared to `jamaicavm`, `jamaicavm_slim` is significantly smaller in size. `jamaicavm_slim` may start up more quickly for small applications, but it will require more time for larger applications. Also, since for `jamaicavm` commonly required standard library classes were pre-compiled and optimized by the Jamaica builder tool (see § 14), `jamaicavm_slim` will perform standard library code more slowly.

13.2.2 jamaicavmm

`jamaicavmm` is the multicore variant of the `jamaicavm_slim`. By using

`jamaicavmm`, you will automatically benefit from the available cores in your machine. Be aware that you need to have an extra license to use this.

`jamaicavmm` accepts the additional command line option `-Xcpus`. See § 13.1.2.

13.2.3 jamaicavmp

`jamaicavmp` is a variant of `jamaicavm_slim` that collects profiling information. This profiling information can be used when creating an optimized version of the application using option `-useProfile` *file* of the Jamaica builder command (see § 14).

The profiling information is written to a file whose name is the name of the main class of the executed Java application with the suffix `.prof`. The following run of the HelloWorld application available in the examples (see § 2.4) shows how the profiling information is written after the execution of the application.

```
> jamaicavmp -cp classes HelloWorld
              Hello        World!
           Hello       World!
        Hello     World!
     Hello    World!
   Hello   World!
 Hello  World!
 [...]
Start writing profile data into file 'HelloWorld.prof'
 Write threads data...
 Write invocation data...
Done writing profile data
```

Profiling information is written after the termination of the application or when Ctrl-C is pressed. As an alternative requesting a profile dump remotely is possible.

For explicit termination, the application needs to be rewritten to terminate at a certain point, e.g., after a timeout or on a certain user input. The easiest means to terminate an application is via a call to `System.exit()`. Otherwise, all threads that are not daemon threads need to be terminated.

To request a remote profile, the property `jamaica.profile_request_port` has to be set to a port number. Then, a request to write the profile can be sent via the command

`jamaicavm com.aicas.jamaica.lang.Profile` *host port*

See § 13.4 for more information on this mechanism.

Profiling information is always appended to the profiling file. This means that profiling information from several profiling runs of the same application, e.g.

using different input data, will automatically be written into a single profiling file. To fully overwrite the profiling information, e.g., after a major change in the application, the profiling file must be deleted manually.

The collection of profiling information requires additional CPU time and memory to store this information. It may therefore be necessary to increase the memory size. Also expect poorer runtime performance during a profiling run.

`jamaicavmp` accepts the following additional command line option.

Option `-XprofileFilename` *filename*

This option selects the name of the file to which the profile data is to be written. If this option is not provided, the default file name is used, consisting of the main class name and the suffix `.prof`.

13.2.4 jamaicavmdi

The `jamaicavmdi` command is a variant of `jamaicavm_slim` that includes support for the JVMTI debugging interface. It includes a debugging agent that can communicate with remote source-level debuggers such as Eclipse.

`jamaicavmdi` accepts the following additional command line option.

Option `-agentlib:`*libname*`[=`*options*`]`

The `agentlib` option loads and runs the dynamic JVMTI agent library *libname* with the given options. Be aware that JVMTI is not yet fully implemented, so not every agent will work. Jamaica comes with a statically built in debugging agent that can be selected by setting `BuiltInAgent` as name. The transport layer must be sockets. A typical example of using this option is

```
-agentlib:BuiltInAgent=transport=dt_socket,server=y,
  suspend=y,address=8000
```

(To be typed in a single line.) This starts the application and waits for an incoming connection of a debugger on port 8000. See § 8.1 for further information on the options that can be provided to the built-in agent for remote debugging.

13.3 Environment Variables

The following environment variables control `jamaicavm` and its variants. The defaults may vary for host and target platforms. The values given here are for guidance only. In order to find out the defaults used by a particular VM, invoke it with command line option `-help`.

CLASSPATH Path list to search for class files.

JAMAICAVM_SCHEDULING_POLICY Native thread scheduling policy on POSIX systems. Setting the scheduling policy may require root access. These are the available values:

- `OTHER` — default scheduling
- `FIFO` — first in first out
- `RR` — round robin

The default is `OTHER`. For obtaining real-time performance, `FIFO` is required. See § 10.5.3 for details.

JAMAICAVM_HEAPSIZE Heap size in bytes, default 2M

JAMAICAVM_MAXHEAPSIZE Max heap size in bytes, default 768M

JAMAICAVM_HEAPSIZEINCREMENT Heap size increment in bytes, default 4M

JAMAICAVM_JAVA_STACKSIZE Java stack size in bytes, default 64K

JAMAICAVM_NATIVE_STACKSIZE Native stack size in bytes, default 150K

JAMAICAVM_NUMTHREADS Initial number of Java threads, default: 10

JAMAICAVM_MAXNUMTHREADS Maximum number of Java threads, default: 511

JAMAICAVM_NUMJNITHREADS Initial number of threads for the JNI function `JNI_AttachCurrentThread`, default: 0

JAMAICAVM_FINALIZERPRI The Java priority of the finalizer thread. This thread executes the `finalize` method of objects before their memory is reclaimed by the GC, default: 10

JAMAICAVM_PRIMAP Priority mapping of Java threads to native threads

JAMAICAVM_TIMESLICE Time slicing for instances of `java.lang.Thread`. See Builder option `timeSlice`.

JAMAICAVM_CONSTGCWORK Amount of garbage collection per block if set to value >0. Amount of garbage collection depending on amount of free memory if set to 0. Stop the world GC if set to -1. Default: 0.

JAMAICAVM_ANALYZE Enable memory analysis mode with a tolerance given in percent (see Builder option `analyze`), default: 0 (disabled).

JAMAICAVM_RESERVEDMEMORY Set the percentage of memory that should be reserved by a low priority thread for fast burst allocation (see Builder option `reservedMemory`), default: 10.

JAMAICAVM_SCOPEDSIZE Size of scoped memory, default: 0

JAMAICAVM_IMMORTALSIZE Size of immortal memory, default: 0

JAMAICAVM_LAZY Use lazy class loading/linkage (1) or load/link all classes at startup (0), default: 1.

JAMAICAVM_STRICTRTSJ Use strictRTSJ rules (1) or relaxed Jamaica rules (0), default: 0

JAMAICAVM_PROFILEFILENAME File name for profile, default: *class*`.prof`, where *class* is the name of the main class.

13.4 Java Properties

A Java property is a string name that has an assigned string value. This section lists Java properties that Jamaica uses in addition to those used by a standard Java implementation. These properties are available with the pre-built VM commands described in this chapter as well as for applications created with the Jamaica Builder.

13.4.1 User-Definable Properties

The standard libraries that are delivered with JamaicaVM can be configured by setting specific Java properties. A property is passed to the Java code via the JamaicaVM option

`-D`*name*`=`*value*

or, when building an application with the Builder, via option

`-XdefineProperty+=`*name*`=`*value*

`jamaica.cost_monitoring_accuracy` = ***num***
This integer property specifies the resolution of the cost monitoring that is used for RTSJ's cost overrun handlers. The accuracy is given in nanoseconds, the default value is `5000000`, i.e., an accuracy of 5ms. The accuracy

specifies the maximum value the actual cost may exceed the given cost budget before a cost overrun handler is fired. A high accuracy (a lower value) causes a higher runtime overhead since more frequent cost budget checking is required. See also § 9.5, Limitations of the RTSJ implementation.

`jamaica.cpu_mhz` = *num*
This integer option specifies the CPU speed of the system JamaicaVM executes on. This number is used on systems that have a CPU cycle counter to measure execution time for the RTSJ's cost monitoring functions. If the CPU speed is not set and it could not be determined from the system (e.g., on Linux via reading file `/proc/cpuinfo`), the CPU speed will be measured on VM startup and a warning will be printed. An example setting for a system running at 1.8GHz would be `-Djamaica.cpu_mhz=1800.0`.

`jamaica.monotonic_currentTimeMillis`
Enable an additional check that enforces that the method `java.lang.System.currentTimeMillis()` always returns a non-negative and monotonically increasing value.

`jamaica.err_to_file`
If a file name is given, all output sent to System.err will be redirected to this file.

`jamaica.err_to_null`
If set to true, all output sent to System.err will be ignored. This is useful for graphical applications if textual output is very slow. The default value for this property is false.

`jamaica.fontproperties` = *resource*
This property specifies the name of a resource that instructs JamaicaVM which fonts to load. The default value is the resource `com/aicas/jamaica/awt/fonts.properties`. The property may be set to a user defined resource file to change the set of supported fonts. The specified file itself is a property file that maps font names to resource file names.

`jamaica.full_stack_trace_on_sig_quit`
If this Boolean property is set, then the default handler for POSIX signal SIGQUIT (`Ctrl-\` on Unix-based platforms) is changed to print full stack trace information in addition to information on thread states, which is the default. See also `jamaica.no_sig_quit_handler`.

`jamaica.gcthread_pri` = *n*
If set to an integer value larger than or equal to `0`, this property instructs the virtual machine to launch a garbage collection thread at the given Java priority. A value of `0` will result in a Java priority `1` with micro adjustment `-1`,

i.e., the scheduler will give preference to other threads running at priority `1`. By default, a GC thread is not used. See § 7.1.5 for more details.

`jamaica.java_thread_default_affinity`
Default affinity set of normal Java threads. E.g., `7,8,9` for CPUs 7, 8 and 9.

`jamaica.heap_so_default_affinity`
Default affinity set of RTSJ schedulable objects (`RealtimeThread` and `AsyncEventHandler`) running in heap memory. E.g., `0,1,2` for CPUs 0, 1 and 2.

`jamaica.loadLibrary_ignore_error`
This property specifies whether every unsuccessful attempt to load a native library dynamically via System.loadLibrary() should be ignored by the VM at runtime. If set to true and System.loadLibrary() fails, no UnsatifiedLinkError will be thrown at runtime. The default value for this property is false.

`jamaica.noheap_so_default_affinity`
Default affinity set of RTSJ schedulable objects (`RealtimeThread` and `AsyncEventHandler`) running in no-heap memory. E.g., `4,5,6` for CPUs 4, 5 and 6.

`jamaica.no_sig_int_handler`
If this boolean property is set, then no default handler for POSIX signal SIGINT (`Ctrl-C` on most platforms) will be created. The default handler that is used when this property is not set prints "`*** break.`" to `System.err` and calls `System.exit(130)`.

`jamaica.no_sig_quit_handler`
If this Boolean property is set, then no default handler for POSIX signal SIGQUIT (`Ctrl-\` on Unix-based platforms) will be created. The default handler that is used when this property is not set prints the current thread states via a call to `com.aicas.jamaica.lang.Debug.dump.ThreadStates()`. See also `jamaica.full_stack_trace_on_sig_quit`.

`jamaica.no_sig_term_handler`
If this boolean property is set, then no default handler for POSIX signal SIGTERM (default signal sent by `kill`) will be created. The default handler that is used when this property is not set prints "`*** terminate.`" to `System.err` and calls `System.exit(130)`.

`jamaica.out_to_file`
If a file name is given, all output sent to System.out will be redirected to this file.

jamaica.out_to_null
: If set to true, all output sent to System.out will be ignored. This is useful for graphical applications if textual output is very slow. The default value for this property is false.

jamaica.profile_groups = ***groups***
: To analyze the application, additional information can be written to the profile file. This can be done by specifying one or more (comma separated) groups with that property. The following groups are currently supported: `builder` (default), `memory`, `speed`, `all`. See § 5 for more details.

jamaica.profile_request_port = ***port***
: When using the profiling version of JamaicaVM (`jamaicavmp` or an application built with "`-profile=true`"), then this property may be set to an integer value larger than `0` to permit an external request to dump the profile information at any point in time. Setting this property to a value larger than `0` also supresses dumping the profile to a file when exiting the application. See § 5 for more details.

jamaica.processing_group_default_affinity
: Default affinity set for RTSJ processing groups (class `ProcessingGroupParameters`). E.g., `10,11` for CPUs 10 and 11.

jamaica.reservation_thread_affinity
: Affinity to be used for memory reservation threads. The cardinality of the given set defines the number of memory reservation threads to be used. E.g., `12,13` to use two memory reservation threads running on CPUs 12 and 13.

jamaica.reservation_thread_priority = ***n***
: If set to an integer value larger than or equal to `0`, this property instructs the virtual machine to run the memory reservation thread at the given Java priority. A value of `0` will result at a Java priority `1` with micro adjustment `-1`, i.e., the scheduler will give preference to other threads running at priority `1`. By default, the priority of the reservation thread is set to `0` (i.e., Java priority 1 with micro adjustment -1). The priority may be followed by a `+` or `-` character to select priority micro-adjustment `+1` or `-1`, respectively. Setting this property, e.g., to `10+` will run the memory reservation thread at a priority higher than all normal Java threads, but lower than all RTSJ threads. See § 7.1.4 for more details.

jamaica.scheduler_events_port
: This property defines the port where the ThreadMonitor can connect to receive scheduler event notifications.

jamaica.scheduler_events_port_blocking
: This property defines the port where the ThreadMonitor can connect to receive scheduler event notifications. The Jamaica runtime system stops before entering the main method and waits for the ThreadMonitor to connect.

jamaica.softref.minfree
: Minimum percentage of free memory for soft references to survive a GC cycle. If the amount of free memory drops below this threshold, soft references may be cleared. In JamaicaVM, the finalizer thread is responsible for clearing soft references. The default value for this property is 10%.

jamaica.x11.display
: This property defines the X11 display to use for X11 graphics. This property takes precedence over a display set via the environment variable DISPLAY.

jamaica.xprof = *n*
: If set to an integer value larger than `0` and less or equal to `1000`, this property enables the `jamaicavm`'s option `-Xprof`. If set, the property's value specifies the number of profiling samples to be taken per second, e.g., `-Djamaica.xprof=100` causes the profiling to make 100 samples per second. See § 13.1.2 for more details.

java.home = *dir*
: The home of the Java runtime environment. When Java standard classes need to locate their associated resources — for example, time zone information — the folder *dir*`/lib` is searched. If the directory exists and the resource is found, it is taken from there, otherwise the resource built into the executable is used.

  The main use of this property is to override resources built into a VM executable. If the property is not set, it is computed based on the location of the VM or application executable. If the executable's parent folder is `bin` the property is set to the parent of the `bin` folder. Otherwise it is set to the parent of the executable. If the parent directory of the executable cannot be determined (lacking operating system functionality) the value of this property is undefined.

13.4.2 Predefined Properties

The JamaicaVM defines a set of additional properties that contain information specific to Jamaica:

jamaica.boot.class.path
: The boot class path used by JamaicaVM. This is not set when a stand-alone application has been built using the Builder (see § 14).

jamaica.buildnumber
The build number of the JamaicaVM.

jamaica.byte_order
One of `BIG_ENDIAN` or `LITTLE_ENDIAN` depending on the endianness of the target system.

jamaica.heapSizeFromEnv
If the initial heap size may be set via an environment variable, this is set to the name of this environment variable.

jamaica.immortalMemorySize
The size of the memory available for immortal memory.

jamaica.maxNumThreadsFromEnv
If the maximum number of threads may be set via an environment variable, this is set to the name of this environment variable.

jamaica.numThreadsFromEnv
If the initial number of threads may be set via an environment variable, this is set to the name of this environment variable.

jamaica.release
The release number of the JamaicaVM.

jamaica.scopedMemorySize
The size of the memory available for scoped memory.

jamaica.strictRTSJ
Boolean property. Value depends on the setting of the `-strictRTSJ` option that was used when building the application. See § 14.2 for more details.

jamaica.version
The version number of the JamaicaVM.

jamaica.word_size
One of `32` or `64` depending on the word size of the target system.

sun.arch.data.model
One of `32` or `64` depending on the word size of the target system.

13.5 Exitcodes

Tab. 13.1 lists the exit codes of the Jamaica VMs. Standard exit codes are exit codes of the application program. Error exit codes indicate an error such as insufficient memory. If you get an exit code of an internal error please contact aicas

Standard exit codes	
0	Normal termination
1	Exception or error in Java program
2..63	Application specific exit code from `System.exit()`
Error codes	
64	JamaicaVM failure
65	VM not initialized
66	Insufficient memory
67	Stack overflow
68	Initialization error
69	Setup failure
70	Clean-up failure
71	Invalid command line arguments
72	No main class
73	`Exec()` failure
Internal errors	
100	Serious error: `HALT` called
101	Internal error
102	Internal test error
103	Function or feature not implemented
104	Exit by signal
105	Unreachable code executed
130	POSIX signal SigInt
143	POSIX signal SigTerm
255	Unexpected termination

Table 13.1: Exitcodes of the Jamaica VMs

support with a full description of the runtime condition or, if available, an example program for which the error occurred.

Chapter 14

The Jamaica Builder

Traditionally, Java applications are stored in a set of Java class files. To run an application, these files are loaded by a virtual machine prior to their execution. This method of execution emphasizes the dynamic nature of Java applications and allows easy replacement or addition of classes to an existing system.

However, in the context of embedded systems, this approach has several disadvantages. An embedded system might not provide the necessary file system device and file system services. Instead, it is preferable to have all files relevant for an application in a single executable file, which may be stored in read only memory (ROM) within an embedded system.

The Builder provides a way to create a single application out of a set of class files and the Jamaica virtual machine.

14.1 How the Builder tool works

Fig. 14.1 illustrates the process of building a Java application and the JamaicaVM into a single executable file. The Builder takes a set of Java class files as input and by default produces a portable C source file which is compiled with a native C compiler to create an object file for the target architecture. The build object file is then linked with the files of the JamaicaVM to create a single executable file that contains all the methods and data necessary to execute the Java program.

14.2 Builder Usage

The builder is a command-line tool. It is named `jamaicabuilder`. A variety of arguments control the work of the Builder tool. The command line syntax is as follows:

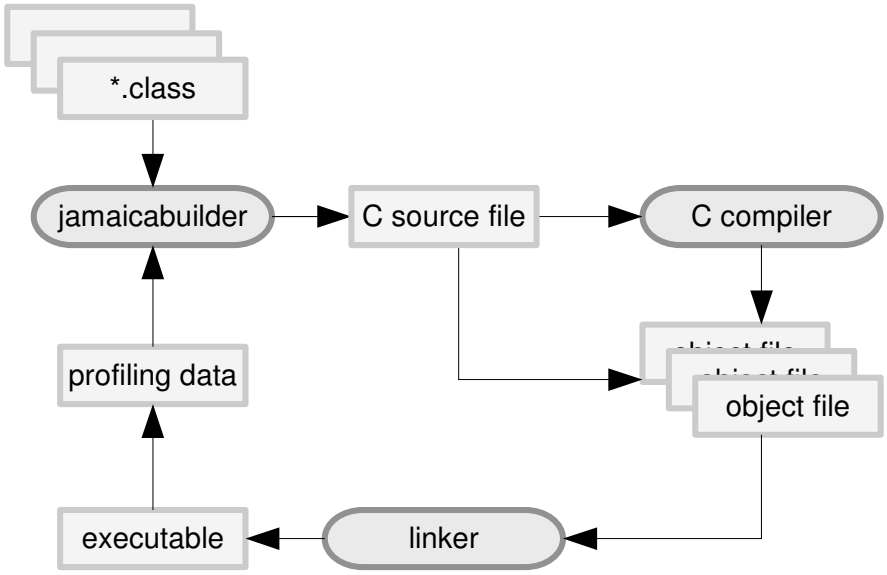


Figure 14.1: The Builder tool

`jamaicabuilder [`*options*`] [`*class*`]`

The builder accepts numerous options for configuring and fine-tuning the created executable. The class argument identifies the main class. It is required unless the main class can be inferred otherwise — for example, from the manifest of a jar file.

The options may be given directly to the Builder via the command line, or by using configuration files. Options given at the command line take priority. Options not specified at the command line are read from configuration files in the following manner:

- The host target is read from *jamaica-home*`/etc/global.conf` and is used as the default target. This file should not contain any other information.
- If the builder option `-configuration` is used, the remaining options are read from the file specified with this option.
- Otherwise *jamaica-home*`/target/`*platform*`/etc/jamaica.conf`, the target-specific configuration file, is used.

The general format for an option is either *−option* for an option without argument or *−option=value* for an option with argument. The following special syntax is accepted:

- To add values to an existing option in the configuration, e.g., `-classpath`, use the following syntax: `-classpath+=`*path*. The value from the configuration is prepended with the value provided on the command line.

- To read values for an option that accepts a list of values, e.g., `-classpath`, from a file instead from the command line or configuration file, use this syntax: `-classpath=@`*file* or `-classpath+=@`*file*. This reads the values from *file* line by line. Empty lines and lines starting with the character "#" (comment) are ignored.

Options that permit lists of arguments can be set by either providing a single list, or by providing an instance of the option for each element of the list. For example, the following are equivalent:

```
-classpath=system_classes:user_classes
-classpath=system_classes -classpath=user_classes
```

The separator for list elements depends on the argument type and is documented for the individual options. As a general rule, paths and file names are separated by the system-specific separator character (colon on Unix systems, semicolon on Windows), for identifiers such as class names and package names the separator is space, and for maps the separator is comma.

If an option's argument contains spaces (for example, a file names with spaces or an argument list) that option must be enclosed in double quotes ("`"`"). The following are well-formed options:

```
"-includeClasses=java.lang... java.util.*"
"-classpath+=system_classes:installation directory"
```

Options that permit a list of mappings as their arguments require one equals sign to start the arguments list and another equals for each mapping in the list.

```
-priMap=1=5,2=7,3=9
```

Default values for many options are target specific. The actual settings may be obtained by invoking the Builder with `-help`. In order to find out the settings for a target other than the host platform, include `-target=`*platform*.

The builder stores intermediate files, in particular generated C and object files, in a temporary folder in the current working directory. For concurrent runs of the builder, in order to avoid conflicts, the builder must be instructed to use distinct temporary directories. In this case, please use the builder option `-tmpdir` to set specifc directories.

14.2.1 General

The following are general options which provide information about the Builder itself or enable the use of script files that specifying further options.

Option `-help (--help, -h, -?)`

The `help` option displays Builder usage and a short description of all possible standard command line options.

Option `-Xhelp (--Xhelp)`

The `Xhelp` option displays Builder usage and a short description of all possible extended command line options. Extended command line options are not needed for normal control of the Builder command. They are used to configure tools and options, and to provide tools required internally for Jamaica VM development.

Option `-agentlib=`*lib=option=val*{`,`*option=val*}

The `agentlib` option loads and runs the dynamic JVMTI agent library *libname* with the given options.

Jamaica comes with a statically built in debugging agent that can be selected by setting `BuiltInAgent` as name. The transport layer must be sockets. A typical example would be: `-agentlib=BuiltInAgent=transport=dt_socket,server=y,suspend=y,address=8000`. This starts the application and waits for an incoming connection of a debugger on port 8000. The `BuiltInAgent` is currently the only agent supported by JamaicaVM.

Option `-version`

Print the version of the Builder and exit.

Option `-verbose=`*n*

The `verbose` option sets the verbosity level for the Builder. If the verbosity level is larger than 0, additional output on the state of the build process is printed to standard out. The maximum level is 7.

The information provided in verbose mode includes: external tools called, warnings, or the list of all class files that are loaded and the methods that are compiled in case compilation is switched on.

Option `-showSettings`

Print the builder settings in property file format. To make these setting the default, replace the file *jamaica-home*`/target/`*platform*`/etc/jamaica.conf` by the output.

Option `-saveSettings=`*file*

If the `saveSettings` option is used, Jamaica Builder options currently in effect are written to the provided file in property file format. To make these setting the default, replace the file *jamaica-home*`/target/`*platform*`/etc/jamaica.conf` by the output.

Option `-configuration=`*file*

The `configuration` option specifies a file to read the set of options used to build the application. The option format must be identical to the one in the default configuration file (*jamaica-home*`/target/`*platform*`/etc/jamaica.conf`). When set, the file *jamaica-home*`/target/`*platform*`/etc/jamaica.conf` is ignored.

14.2.2 Classes, files and paths

These options allow to specify classes and paths to be used by the builder.

Option `-classpath (-cp)[+]=`*classpath*

The `classpath` option specifies the paths that are used to search for class files. A list of paths separated by the path separator char (':' on Unix systems, ';' on Windows) can be specified. This list will be traversed from left to right when the Builder tries to load a class.

Option `-enableassertions (-ea)`

The `enableassertions` option enables assertions for all classes. Assertions are disabled by default.

Option `-main=`*class*

The `main` option specifies the main class of the application that is to be built. This class must contain a static method `void main(String[] args)`. This method is the main entry point of the Java application.

If the `main` option is not specified, the first class of the classes list that is provided to the Builder is used as the main class.

Option `-jar=`*file*

The `jar` option specifies a JAR file with an application that is to be built. This JAR file must contain a `MANIFEST` with a Main-Class entry.

Option `-includeClasses[+]="`*class*`|`*package*{ *class*`|` *package*}`"`

The `includeClasses` option forces the inclusion of the listed classes and packages into the created application. The listed classes with all their methods and fields will be included. This is useful or even necessary if you use reflection with these classes.

Arguments for this option can be: a class name to include the class with all methods and fields, a package name followed by an asterisk to include all classes in the package or a package name followed by "`...`" to include all classes in the package and in all sub-packages of this package.

Example:

```
-includeClasses="java.beans.XMLEncoder java.util.*
                 java.lang..."
```

includes the class `java.beans.XMLEncoder`, all classes in `java.util` and all classes in the package `java.lang` and in all sub-packages of `java.lang` such as `java.lang.ref`.

! The `includeClasses` option affects only the listed classes themselves. Subclasses of these classes remain subject to smart linking.

! From a Unix shell, when specifying an inner class, the dollar sign must be preceded by backslash. Otherwise the shell interprets the class name as an environment variable.

Option `-excludeClasses[+]="`*class*`|`*package*{ *class*`|` *package*}`"`

The `excludeClasses` option forces exclusion of the listed classes and packages from the created application. The listed classes with all their methods and fields will be excluded, even if they were previously included using `includeJAR` or `includeClasses`. This is useful if you want to load classes at runtime.

Arguments for this option can be: a class name to exclude the class with all methods and fields, a package name followed by an asterisk to exclude all classes

in the package or a package name followed by "`...`" to exclude all classes in the package and in all sub-packages of this package.

Example:

```
-excludeClasses="java.beans.XMLEncoder java.util.*
                 java.lang..."
```

excludes the class `java.beans.XMLEncoder`, all classes in `java.util` and all classes in the package `java.lang` and in all sub-packages of `java.lang` such as `java.lang.ref`.

The `excludeClasses` option affects only the listed classes themselves.

Option `-includeInCompile[+]=`*class*`|`*method*{ *class*`|` *method*}

The `includeInCompile` option forces the compilation of the listed methods (when not excluded from the application by the smart linker or by any other means). Arguments must be separated by spaces and enclosed in double quotes (`"`).

Here are examples of arguments: `*` for compiling all classes of the default package, `com.user.*` for compiling all methods of the classes of the package `com.user` excluding subpackages, `com.user...` for compiling all methods of the classes of the package `com.user` including all subpackages, `com.user.MyClass` for compiling all methods of the class `com.user.MyClass`, `com.user.MyClass.add` for compiling all methods named `add` of the class `com.user.MyClass`, and `com.user.MyClass.add(I)V` for compiling the given method.

Option `-excludeFromCompile[+]=`*class*`|`*method*{ *class*`|` *method*}

The `excludeFromCompile` option disables the compilation of the listed methods. Arguments must be separated by spaces and enclosed in double quotes (`"`).

Here are examples of arguments: `*` for disabling the compilation of any method of the default package classes, `com.user.*` for disabling the compilation of any method of the classes of the package `com.user` excluding subpackages, `com.user...` for disabling the compilation of any method from classes of the package `com.user` including subpackages, `com.user.MyClass` for disabling the compilation of methods of the class `com.user.MyClass`, `com.user.MyClass.add` for disabling the compilation of any method named `add` of the class `com.user.MyClass`, and `com.user.MyClass.add(I)V` for disabling the compilation of the given method.

Option `-includeJAR[+]=`*file*{`:`*file*}

The `includeJAR` option forces the inclusion of all classes and all resources contained in the specified files. Any archive listed here must be in the classpath or in the bootclasspath. If a class needs to be included, the implementation in the `includeJAR` file will not necessarily be used. Instead, the first implementation of this class which is found in the classpath will be used. This is to ensure the application behaves in the same way as it would if it were called with the `jamaicavm` or `java` command.

Despite its name, the option accepts directories as well. Multiple archives (or directories) should be separated by the system specific path separator: colon "`:`" on Unix systems and semicolon "`;`" on Windows.

Option `-excludeJAR[+]=`*file*{`:`*file*}

The `excludeJAR` option forces the exclusion of all classes and resources contained in the specified files. Any class and resource found will be excluded from the created application. Use this option to load an entire archive at runtime.

Despite its name, the option accepts directories as well. Multiple archives (or directories) should be separated by the system specific path separator: colon "`:`" on Unix systems and semicolon "`;`" on Windows.

Option `-lazy`

Like other VMs, Jamaica VM by default loads classes dynamically, that is, when they are first used. Since this can cause unpredictable delays during byte code execution, which are not acceptable in real-time environments, this option enables changing this behaviour, by setting `-lazy=false`.

Disabling this option will cause all classes that are referenced from the main class to be loaded before execution of the application starts. This early loading and linking of classes ensures that during the execution of the Java application, no further class loading and linking will be required, ensuring the predictable execution of statements that can cause class loading in a traditional Java environment.

The statements that may cause loading and linking of classes are the same statements that may cause the execution of static initializers of a class: calls of static methods, accesses to static fields and the creation of an instance of a class.

Disabling this option my result in significantly increased startup times of Jamaica VM.

Option `-lazyFromEnv=`*var*

This option causes the creation of an application that reads its `lazy` setting from the specified environment variable. If this variable is not set, the value of boolean option `lazy` will be used. The value of the environment variable must be `0` (for `-lazy=false`) or `1` (for `-lazy=true`).

Option `-destination (-o)=`*name*

The `destination` option specifies the name of the destination executable to be generated by the Builder. If this option is not present, the name of the main class is used as the name of the destination executable.

The destination name can be a path into a different directory. E.g.,

```
-destination myproject/bin/application
```

may be used to save the created executable `application` in `myproject/bin`.

Option `-tmpdir=`*name*

The `tmpdir` option may be used to specify the name of the directory used for temporary files generated by the Builder (such as C source and object files for compiled methods).

Option `-resource[+]=`*name*{`:`*name*}

This option causes the inclusion of additional resources in the created application. A resource is additional data (such as image files, sound files etc.) that can be accessed by the Java application. Within the Java application, the resource data can be accessed using the resource name specified as an argument to `resource`. To load the resource, a call to `Class.getResourceAsStream(`*name*`)` can be used.

If a resource is supposed to be in a certain package, the resource name must include the package name. Any '.' must be replaced by '/'. E.g., the resource `ABC` from package `foo.bar` can be added using `-resource foo/bar/ABC`.

The Builder uses the class path provided through the option `classpath` to search for resources. Any path containing resources that are provided using `resource` must therefore be added to the path provided to `classpath`.

This option expects a list of resource files that are separated using the platform dependent path separator character (e.g., ':').

Option `-setFonts="`*font*{ *font*}`"`

The `setFonts` option can be used to choose the set of TrueType fonts to be included in the target application. The font families `sans`, `serif`, `mono` are supported. If no fonts are required, it can be set to `none`. To use TrueType fonts, a graphics system must be set.

Option `-setGraphics=`*system*

The `setGraphics` option can be used to set the graphics system used by the target application. If no graphics is required, it can be set to `none`.

To get a list of all possible values, invoke the Builder with `-help`.

Option `-setLocalCryptoPolicy=`*policy*

The `setLocalCryptoPolicy` option can be used to set the local crypto policy to be used by the target application. For strong encryption support, this should be set to `unlimited_local_policy.jar`. Please note that strong encryption requires the file *jamaica-home*`/target/`*platform*`/lib/crypto/unlimited_local_policy.jar` to be present, which is not part of a standard Jamaica installation.

Option `-setLocales="`*locale*{ *locale*}`"`

The `setLocales` option can be used to choose the set of locales to be included in the target application. This involves date, currency and number formats. Locales are specified by a lower-case, two-letter code as defined by ISO-639.

Example: `-setLocales="de en"` will include German and English language resources. All country information of those locales, e.g. Swiss currency, will also be included.

To get a list of all possible values, invoke the Builder with `-help`.

Option `-setTimeZones[+]="`*timezone*{ *timezone*}`"`

The `setTimeZones` option can be used to choose the set of time zones to be included in the target application. By default all time zones are built in.

Examples: `-setTimeZones=Europe/Berlin` will include the time zone of Berlin only, `-setTimeZones=Europe` will include all European time zones, `-setTimeZones="Europe/Berlin America/Detroit"` includes time zones for Berlin and Detroit.

See the folder *jamaica-home*`/target/`*platform*`/lib/zi` for the available time zones.

Option `-setProtocols="`*protocol*{ *protocol*}`"`

The `setProtocols` option can be used to choose the set of protocols to be included in the target application.

Example: `-setProtocols="ftp http"` will include handlers for FTP and HTTP protocols.

To get a list of all possible values, invoke the Builder with `-help`.

Option `-incrementalCompilation`

If the `incrementalCompilation` option is set to `true`, C code is split into several files for incremental compilation. The generated code is split into one file per package. This is the default behavior. If this option is set to false, all C code is put into one single, potentially large C source file.

14.2.3 Smart linking

Smart linking and compaction are techniques to reduce the code size and heap memory required by the generated application. These techniques are controlled by the following options.

Option `-smart`

If the `smart` option is set, smart linking takes place at the level of fields and methods. That is, unused fields and methods are removed from the generated code. Otherwise smart linking may only exclude unused classes as a whole. Setting `smart` can result in smaller binary files, smaller memory usage and faster code execution. This option is enabled by default.

Smart linking at the level of fields and methods may not be used for applications that use Java's reflection API (including reflection via the Java Native Interface JNI) to load classes that are unknown at buildtime and therefore cannot be included into the application. This is, for example, the case for classes, which are loaded from a web server at runtime. In such situations, use `-smart=false` to disable smart linking.

Classes loaded via reflection that are known at buildtime should be included via builder options `includeClasses` or `includeJAR`. These options selectively disable smart linking for the included classes.

Option `-closed`

For an application that is `closed`, i.e., that does not load any classes dynamically that are not built into the application by the Builder, additional optimization may

be performed by the Builder and the static compiler. These optimizations cause incorrect execution semantics when additional classes will be added dynamically. Setting option `closed` to true enables such optimizations, a significant enhancement of the performance of compiled code is usually the result.

The additional optimization performed when `closed` is set include static binding of virtual method calls for methods that are not redefined by any of the classes built into the application. The overhead of dynamic binding is removed and even inlining of a virtual method call becomes possible, which often results in even further possibilities for optimizations.

Note that care is needed for an open application that uses dynamic loading even when `closed` is not set. For an open application, it has to be ensured that all classes that should be available for dynamically loaded code need to be included fully using option `includeClasses` or `includeJAR`. Otherwise, the Builder may omit these classes (if they are not referenced by the built-in application), or it may omit parts of these classes (certain methods or fields) that happen not to be used by the built-in application.

Option `-showIncludedFeatures`

The `showIncludedFeatures` option causes the Builder to display the list of classes, methods, fields and resources that were included in the target application. This option can help identify the features that were removed from the target application through mechanisms such as smart linking.

The output of this option consists of lines starting with the string `INCLUDED CLASS`, `INCLUDED METHOD`, `INCLUDED FIELD` or `INCLUDED RESOURCE` followed by the name of the class, method, field or resource. For methods, the signature is shown as well.

Option `-showExcludedFeatures`

The `showExcludedFeatures` option causes the Builder to display the list of methods and fields that were removed from the target application through mechanisms such as smart linking. Only methods and fields from classes present in the built application will be displayed. Used in conjunction with `includeClasses`, `excludeClasses`, `includeJAR` and `excludeJAR` this can help identify which classes were included only partially.

The output of this option consists of lines starting with the string `EXCLUDED METHOD` or `EXCLUDED FIELD` followed by the name and signature of a method or field, respectively.

Option -showNumberOfBlocks

The `showNumberOfBlocks` option causes the Builder to display a table with the number of block needed by all the classes included in the target application.This option can help to calculate the worst case allocation time.

The output of this option consists of a two columns table. The first column is named `Class:` and the second is named `Blocks:`. Next lines contain the name of each class and the corresponding number of blocks.

14.2.4 Profiling and compilation

By default, the builder compiles all application classes and a predefined set of the system classes. Profiling and compilation options enable to fine tune the compilation process for optimal runtime performance of applications generated with the builder.

Option -interpret (-Xint)

The `interpret` option disables compilation of the application. This results in a smaller application and in faster build times, but it causes a significant slow down of the runtime performance.

If none of the options `interpret`, `compile`, or `useProfile` is specified, then the default compilation will be used. The default means that a pregenerated profile will be used for the system classes, and all application classes will be compiled fully. This default usually results in good performance for small applications, but it causes extreme code size increase for larger applications and it results in slow execution of applications that use the system classes in a way different than recorded in the system profile.

Option -compile

The `compile` option enables static compilation for the created application. All methods of the application are compiled into native code causing a significant speedup at runtime compared to the interpreted code that is executed by the virtual machine. Use compilation whenever execution time is important. However, it is often sufficient to compile about 10 percent of the classes, which results in much smaller executables of comparable speed. You can achieve this by using the options `profile` and `useProfile` instead of `compile`. For a tutorial on profiling see Section Performance Optimization in the user manual.

Option `-profile`

The `profile` option builds an application that collects information on the amount of run time spent for the execution of different methods. This information is dumped to a file after a test run of the application has been performed. The name of the file is derived from the name of the executable given via the `destination` option. Alternatively, it may be given with the option `XprofileFilename`.

The information collected in a profiling run can then be used as an input for the option `useProfile` to guide the compilation process. For a tutorial on profiling see Section Performance Optimization in the user manual.

Option `-useProfile[+]=`*file*{`:`*file*}

The `useProfile` option instructs the builder to use profiling information collected using `profile` to restrict compilation to those methods that were most frequently executed during the profiling run. The percentage of methods to be compiled is 10 by default, unless `percentageCompiled` is set to a different value. For a tutorial on profiling see Section Performance Optimization in the user manual.

It is possible to use this option in combination with the option `profile`. This may be useful when the fully interpreted application is too slow to obtain a meaningful profile. In such a case one may achieve sufficient speed up through an initial profile, and use the profiled application to obtain a more precise profile for the final build.

Multiple profiles should be separated by the system specific path separator: colon "`:`" on Unix systems and semicolon "`;`" on Windows.

Option `-percentageCompiled=`*n*

Use profiling information collected using `profile` to restrict compilation to those methods that were most frequently executed during the profiling run. The percentage of methods that are to be compiled is given as an argument to the option `percentageCompiled`. It must be between 0 and 100. Selecting 100 causes compilation of all methods executed during the profiling run, i.e., methods that were not called during profiling will not be compiled.

Option `-inline=`*n*

When methods are compiled (via one of the options `compile`, `useProfile`, or `interpret=false`), this option can be used to set the level of inlining to be used by the compiler. Inlining typically causes a significant speedup at runtime since the overhead of performing method calls is avoided. Nevertheless, inlining

causes duplication of code and hence might increase the binary size of the application. In systems with tight memory resources, inlining may therefore not be acceptable

Eleven levels of inlining are supported by the Jamaica compiler ranging from 0 (no inlining) to 10 (aggressive inlining).

Option `-optimize (-optimise)=`*type*

The `optimize` option enables to specify optimizations for the compilation of intermediate C code to native code in a platform independent manner, where *type* is one of `none`, `size`, `speed`, and `all`. The optimization flags only affect the C compiler, and they are only given to it if the application is compiled without the `debug` option.

Option `-target=`*platform*

The `target` option specifies a target platform. For a list of all available platforms of your Jamaica VM Distribution, use `XavailableTargets`.

14.2.5 Heap and stack configuration

Configuring heap and stack memory has an important impact not only on the amount of memory required by the application but on the runtime performance and the realtime characteristics of the code as well. The Jamaica Builder therefore provides a number of options to configure heap memory and stack available to threads.

Option `-heapSize=`*n*`[K|M]`

The `heapSize` option sets the heap size to the specified size given in bytes. The heap is allocated at startup of the application. It is used for static global information (such as the internal state of the Jamaica Virtual Machine) and for the garbage collected Java heap.

The heap size may be succeeded by the letter 'K' or 'M' to specify a size in KBytes (1024 bytes) or MBytes (1048576 bytes). The minimum required heap size for a given application can be determined using option `analyze`.

Option `-maxHeapSize=`*n*`[K|M]`

The `maxHeapSize` option sets the maximum heap size to the specified size given in bytes. If themaximum heap size is larger than the heap size, the heap size will be increased dynamically on demand.

The maximum heap size may be succeeded by the letter 'K' or 'M' to specify a size in KBytes (1024 bytes) or MBytes (1048576 bytes). The minimum value is 0 (for no dynamic heap size increase).

Option `-heapSizeIncrement=`*n*`[K|M]`

The `heapSizeIncrement` option specifies the steps by which the heap size can be increased when the maximum heap size is larger than the heap size.

The maximum heap size may be succeeded by the letter 'K' or 'M' to specify a size in KBytes (1024 bytes) or MBytes (1048576 bytes). The minimum value is 64k.

Option `-javaStackSize=`*n*`[K|M]`

The `javaStackSize` option sets the stack size to be used for the Java runtime stacks of all Java threads in the built application. Each Java thread has its own stack which is allocated from the global Java heap. The stack size consequently has an important impact on the heap memory required by an application. A small stack size is recommended for systems with tight memory constraints. If the stack size is too small for the application to run, a stack overflow will occur and a corresponding error reported.

The stack size may be followed by the letter 'K' or 'M' to specify a size in KBytes (1024 bytes) or MBytes (1048576 bytes). The minimum stack size is 1k.

Option `-nativeStackSize=`*n*`[K|M]`

The `nativeStackSize` option sets the stack size to be used for the native runtime stacks of all Java threads in the built application. Each Java thread has its own native stack. Depending on the target system, the stack is either allocated and managed by the underlying operating system, as in many Unix systems, or allocated from the global heap, as in some small embedded systems. When native stacks are allocated from the global heap, stack size consequently has an important impact on the heap memory required by an application. A small stack size is recommended for systems with tight memory constraints. If the selected stack size is too small, an error may not be reported because the stack-usage of native code may cause a critical failure.

For some target systems, like many Unix systems, a stack size of 0 can be selected, meaning "unlimited". In that case the stack size is increased dynamically as needed.

The stack size may be followed by the letter 'K' or 'M' to specify a size in KBytes (1024 bytes) or MBytes (1048576 bytes). The minimum stack size is 1k

if not set to 'unlimited' (value of 0).

Option `-heapSizeFromEnv=`*var*

The `heapSizeFromEnv` option enables the application to read its heap size from the specified environment variable. If this variable is not set, the heap size specified using `-heapSize` *n* will be used.

Option `-maxHeapSizeFromEnv=`*var*

The `maxHeapSizeFromEnv` option enables the application to read its maximum heap size from the specified environment variable. If this variable is not set, the maximum heap size specified using `-maxHeapSize` *n* will be used.

Option `-heapSizeIncrementFromEnv=`*var*

The `heapSizeIncrementFromEnv` option enables the application to read its heap size increment from the specified environment variable within. If this variable is not set, the heap size increment specified using `-heapSizeIncrement` *n* will be used.

Option `-javaStackSizeFromEnv=`*var*

The `javaStackSizeFromEnv` option enables the application to read its Java stack size from the specified environment variable. If this variable is not set, the stack size specified using `-javaStackSize` *n* will be used.

Option `-nativeStackSizeFromEnv=`*var*

The `nativeStackSizeFromEnv` option enables the application to read its native stack size from the specified environment variable. If this variable is not set, the stack size specified using `-nativeStackSize` *n* will be used.

14.2.6 Threads, priorities and scheduling

Configuring threads has an important impact not only on the runtime performance and realtime characteristics of the code but also on the memory required by the application. The Jamaica Builder provides a range of options for configuring the number of threads available to an application, priorities and scheduling policies.

Option -numThreads=*n*

The `numThreads` option specifies the initial number of Java threads supported by the destination application. These threads and their runtime stacks are generated at startup of the application. A large number of threads consequently may require a significant amount of memory.

The minimum number of threads is two, one thread for the main Java thread and one thread for the finalizer thread.

Option -maxNumThreads=*n*

The `maxNumThreads` options specifies the maximum number of Java threads supported by the destination application. If the maximum number of threads is larger than the values specified for `numThreads`, threads will be added dynamically.

Adding new threads requires unfragmented heap memory. It is strongly recommended to use -maxNumThreads only in conjunction with `maxHeapSize` set to a value larger than `heapSize`. This will permit the VM to increase the heap when memory is fragmented.

The absolute maximum number of threads for the Jamaica VM is 511.

Option -numJniAttachableThreads=*n*

The `numJniAttachableThreads` specifies the initial number of Java thread structures that will be allocated and reserved for calls to the JNI Invocation API functions. These are the functions `JNI_AttachCurrentThread` and `JNI_AttachCurrentThreadAsDaemon`. These threads will be allocated on VM startup, such that no additional allocation is required on a later call to `JNI_AttachCurrentThread` or `JNI_AttachCurrentThreadAsDaemon`.

Even if this option is set to zero, it still will be possible to use these functions. However, then these threads will be allocated dynamically when needed.

Since non-fragmented memory is required for the allocation of these threads, a later allocation may require heap expansion or may fail due to fragmented memory. It is therefore recommended to pre-allocate these threads.

The number of JNI attachable threads that will be required is the number of threads that will be attached simultaneously. Any thread structure that will be detached via `JNI_DetachCurrentThread` will become available again and can be used by a different thread that calls `JNI_AttachCurrentThread` or `JNI_AttachCurrentThreadAsDaemon`.

Option `-threadPreemption=`*n*

Compiled code contains special instructions that permit thread preemption. These instructions have to be executed often enough to allow a thread preemption time that is sufficient for the destination application. As the instructions cause an overhead in code size and runtime performance one would want to generate this code as rarely as possible.

The `threadPreemption` option enables setting of the maximum number of intermediate instructions that are permitted between the execution of thread preemption code. This directly affects the maximum thread preemption time of the application. One intermediate instruction typically corresponds to 1-2 machine instructions. There are some intermediate instructions (calls, array accesses) that can be more expensive (20-50 machine instructions).

The thread preemption must be at least 10 intermediate instructions.

Option `-timeSlice=`*n*

For thread instances of `java.lang.Thread` of equal priority, round robin scheduling is used when several threads are running simultaneously. Using the `timeSlice` option, the maximum size of such a time slice can be given in nanoseconds. A special synchronization thread is used that waits for the length of a time slice and permits thread switching after every slice. Time slicing does not affect real-time threads.

If no round robin scheduling is needed for threads of equal priority, the size of the time slice may be set to zero. In this case, the synchronization thread is not required, so fewer system resources are needed.

Option `-timeSliceFromEnv=`*var*

The `timeSliceFromEnv` option creates an application that reads the time slice settings for instances of `java.lang.Thread` from the environment variable *var*. If this variable is not set, the mapping specified using `-timeSlice` *n* will be used.

Option `-finalizerPri=`*pri*

The `finalizerPri` option sets the Java priority of the finalizer thread to *pri*. The finalizer thread is a daemon thread that runs in the background and executes the method `finalize()` of objects that are about to be freed by the garbage collector. The memory of objects that have such a method cannot be freed before this method is executed.

If the finalizer thread priority is set to zero, no finalizer thread will be created. In that case, the memory of objects that are found to be unreachable by the garbage collector cannot be freed before their finalizers are executed explicitly by calling `java.lang.Runtime.runFinalization()`. It can be useful on very small systems not to use a finalizer thread. This reduces the use of system resources. The priority must be one of the Java priorities 1 through 10 (corresponding to the ten priority levels of `java.lang.Thread`).

Option `-numThreadsFromEnv=`*var*

The `numThreadsFromEnv` option enables the application to read the number of threads from the specified environment variable. If this variable is not set, the number specified using `-numThreads` *n* will be used.

Option `-maxNumThreadsFromEnv=`*var*

The `maxNumThreadsFromEnv` option enables the application to read the maximum number of threads from the environment variable specified within. If this variable is not set, the number specified using `-maxNumThreads` *n* will be used.

Option `-numJniAttachableThreadsFromEnv=`*var*

The `numJniAttachableThreadsFromEnv` option enables the application to read its initial number of JNI attachable threads from the environment variable specified within. If this variable is not set, the value specified using the option `-numJniAttachableThreads` *n* will be used.

Option `-finalizerPriFromEnv=`*var*

The `finalizerPriFromEnv` option enables the application to read its finalizer priority from the environment variable specified within. If this variable is not set, the finalizer priority specified using `finalizerPri` *n* will be used.

Option `-priMap[+]=`*jp*`=`*sp*{`,`*jp*`=`*sp*}

The `priMap` option defines the mapping of priority levels of Java threads to native priorities of system threads. This map is required since JamaicaVM implements Java threads as operating system threads.

The Java thread priorities are integer values in the range 0 through 127, where 0 corresponds to the lowest priority and 127 to the highest priority. Not all Java

thread priorities are normally used by an application, and therefore not all priorities need to be mapped to system priorities. Mappings for the priority levels of `java.lang.Thread` (ranging from 1 through 10) and the priority levels of `javax.realtime.RealtimeThread` (ranging from 11 through 38) and the synchronization thread (priority level 39 or higher) must be provided. The synchronization thread always takes the highest available Java priority. Its purpose is to provide round robin scheduling for instances of `java.lang.Thread` of equal priority. The Java priority level 0 is optional, it may be used to provide a specific native priority for Java priority level 1 with micro-adjustment -1 (see class `com.aicas.jamaica.lang.Scheduler`).

Each Java priority level from 1 up to the maximal used priority must be mapped to a system priority, and the mapping must be monotonic. That is, a higher Java priority level may not be mapped to a lower system priority. The only exception is the priority of the synchronization thread, which may be mapped to any system priority. To simplify the notation, a range of priority levels or system priorities can be described using the notation *from*`..`*to*.

Example 1: `-priMap=1..10=5,11..38=7..34,39=6` will cause all normal threads to use system priority 5, while the real-time threads will be mapped to priorities 7 through 34. The synchronization thread will use priority 6. There will be 28 priority levels for instances of `RealtimeThread`, and the synchronization thread will run at a system priority lower than the real-time threads.

Example 2: `-priMap=1..51=102..2` on a system where higher priorities are denoted by smaller numbers will cause the use of system priorities 102, 100, 98 through 4 for priority levels 1 through 51. The synchronization thread will use priority 2. There will be 40 priority levels available for instances of `RealtimeThread`.

The default of this option is system specific. It maps at least the Java priority levels required for `java.lang.Thread` and `RealtimeThread`, and for the synchronization thread to suitable system priorities.

Note: If round robin scheduling is not needed for instances of `java.lang.Thread` of equal priority and the timeslice is set to zero (`-timeSlice=0`), the synchronization thread is not required and no system priority needs to be given for it.

Option `-priMapFromEnv=`*var*

The `priMapFromEnv` option creates an application that reads the priority mapping of Java threads to native threads from the environment variable *var*. If this variable is not set, the mapping specified using `-priMap` *jp*`=`*sp*{`,`*jp*`=`*sp*} will be used.

Option `-schedulingPolicy=`*schedulingPolicy*

The `schedulingPolicy` option sets the thread scheduling policy. Examples include OTHER, FIFO, or RR.

Option `-schedulingPolicyFromEnv=`*var*

The `schedulingPolicy` option enables the application to read its scheduling policy from the specified environment variable. If this variable is not set, the scheduling policy specified using `-schedulingPolicy` *n* will be used.

14.2.7 Parallel Execution

The parallel version of JamaicaVM can execute several threads, including the garbage collection, in parallel and therefore improves the runtime performance when using multicore systems. Notice that you need to have an extra license to use the parallel version of JamaicaVM.

Option `-parallel`

The `parallel` option instructs the Builder to create an application that can make use of several processors executing Java code in parallel.

14.2.8 GC configuration

The following options provide ways to analyze the application's memory demand and to use this information to configure the garbage collector for the desired real-time behavior.

Option `-analyze (-analyse)=`*tolerance*

The `analyze` option enables memory analyze mode with tolerance given in percent. In memory analyze mode, the memory required by the application during execution is determined. The result is an upper bound for the actual memory required during a test run of the application. This bound is at most the specified tolerance larger than the actual amount of memory used during runtime.

The result of a test run of an application built using `analyze` can then be used to estimate and configure the heap size of an application such that the garbage collection work that is performed on an allocation never exceeds the amount allowed to ensure timely execution of the application's realtime code.

Using `analyze` can cause a significant slowdown of the application. The application slows down as the tolerance is reduced, i.e., the lower the value specified as an argument to `analyze`, the slower the application will run.

In order to configure the application heap, a version of the application must be built using the option `analyze` and, in addition, the exact list of arguments used for the final version. The heap size determined in a test run can then be used to build a final version using the preferred heap size with desired garbage collection overhead. To reiterate, the argument list provided to the Builder for this final version must be the same as the argument list for the version used to analyze the memory requirements. Only the `heapSize` option of the final version must be set accordingly and the final version must be built without setting `analyze`.

Option `-analyzeFromEnv (-analyseFromEnv)=`*var*

The `analyzeFromEnv` option enables the application to read the amount of analyze accuracy of the garbage collector from the environment variable specified within. If this variable is not set, the value specified using `-analyze` *n* will be used. Setting the environment variable to '0' will disable the analysis and cause the garbage collector to use dynamic garbage collection mode.

Option `-constGCwork=`*n*

The `constGCwork` option runs the garbage collector in static mode. In static mode, for every unit of allocation, a constant number of units of garbage collection work is performed. This results in a lower worst case execution time for the garbage collection work and allocation and more predictable behavior, compared with dynamic mode, because the amount of garbage collection work is the same for any allocation. However, static mode causes higher average garbage collection overhead compared to dynamic mode.

The value specified is the number for units of garbage collection work to be performed for a unit of memory that is allocated. This value can be determined using a test run built with -analyze set.

A value of '0' for this option chooses the dynamic GC work determination that is the default for Jamaica VM.

A value of '-1' enables a stop-the-world GC, see option `stopTheWorldGC` for more information.

A value of '-2' enables an atomic GC, see option `atomicGC` for more information.

The default setting chooses dynamic GC: the amount of garbage collection work on an allocation is then determined dynamically depending on the amount of free memory.

Option `-constGCworkFromEnv=`*var*

The `constGCworkFromEnv` option enables the application to read the amount of static garbage collection work on an allocation from the environment variable specified within. If this variable is not set, the value specified with the option `-constGCwork` will be used.

Option `-stopTheWorldGC`

The `stopTheWorlsGC` option enables blocking GC, i.e., no GC activity is performed until the heap is fully filled. Only then, a complete GC cycle is performed at once, causing a potentially long pause for the application. During this GC cycle, any thread that performs heap memory allocation will be blocked, but threads that do not perform heap allocation may continue to run.

If stop-the-world GC is enabled via this option, even `RealtimeThreads` and `NoHeapRealtimeThreads` may be blocked by GC activity if they allocate heap memory. `RealtimeThreads` and `NoHeapRealtimeThreads` that run in `ScopedMemory` or `ImmortalMemory` will not be stopped by the GC

A stop-the-world GC enables a higher average throughput compared to incremental GC, but at the cost of losing realtime behaviour for all threads that perform heap allocation.

Option `-atomicGC`

The `atomicGC` option enables atomic GC, i.e., no GC activity is performed until the heap is fully filled. Only then, a complete GC cycle is performed at once, causing a potentially long pause for the application. During this GC cycle, all Java threads will be blocked.

When this option is set, even `NoHeapRealtimeThreads` will be stopped by GC work, so all realtime guarantees will be lost!

This mode permits more efficient code compared to `stopTheWorldGC` since it disables certain tracking code (write barriers) that is required for the incremental GC.

Option `-reservedMemory=`*percentage*

Jamaica VM's realtime garbage collector performs GC work at allocation time. This may reduce the responsiveness of applications that have long pause times with little or no activity and are preempted by sudden activities that require a burst of memory allocation. The responsiveness of such burst allocations can be improved significantly via reserved memory.

If the `reservedMemory` option is set to a value larger 0, then a low priority thread will be created that continuously tries to reserve memory up to the percentage of the total heap size that is selected via this option. Any thread that performs memory allocation will then use this reserved memory to satisfy its allocations whenever there is reserved memory available. For these allocations of reserved memory, no GC work needs to be performed since the low priority reservation thread has done this work already. Only when the reserved memory is exhausted will GC work to allow further allocations be performed.

The overall effect is that a burst of allocations up to the amount of reserved memory followed by a pause in activity that was long enough during this allocation will require no GC work to perform the allocation. However, any thread that performs more allocation than the amount of memory that is currently reserved will fall back to the performing GC work at allocation time.

The disadvantage of using reserved memory is that the worst-case GC work that is required per unit of allocation increases as the size of reserved memory is increased. For a detailed output of the effect of using reserved memory, run the application with option `-analyze` set together with the desired value of reserved memory.

Option `-reservedMemoryFromEnv=`*var*

The `reservedMemoryFromEnv` option enables the application to read the percentage of reserved memory from the environment variable specified within. If this variable is not set, the value specified using `-reservedMemory` *n* will be used. See option `reservedMemory` for more information on the effect of this option.

14.2.9 RTSJ settings

The following options set values that are relevant for the Real-Time Specification for Java extensions through classes javax.realtime.* that are provided by JamaicaVM.

Option `-strictRTSJ`

The Real-Time Specification for Java (RTSJ) defines a number of classes in the package `javax.realtime`. These classes can be used to create realtime threads with stricter semantics than normal Java threads. In particular, these threads can run in their own memory areas (scoped memory) that are not part of the Java heap, such that memory allocation is independent of garbage collector intervention. It is even possible to create threads of class `javax.realtime.`

`NoHeapRealtimeThread` that may not access any objects stored on the Java heap.

In Jamaica VM, normal Java Threads do not suffer from these restrictions. Priorities of normal threads may be in the range permitted for `RealtimeThreads` (see option `priMap`). Furthermore, any thread may access objects allocated on the heap without having to fear being delayed by the garbage collector. Any thread is safe from being interrupted or delayed by garbage collector activity. Only higher priority threads can interrupt lower priority threads.

When using Jamaica VM, it is thus not necessary to use non-heap memory areas for realtime tasks. It is possible for any thread to access objects on the heap. Furthermore, scoped memory provided by the classes defined in the RTSJ are available to normal threads as well.

The strict semantics of the RTSJ require a significant runtime overhead to check that an access to an object is legal. Since these checks are not needed by Jamaica VM, they are disabled by default. However, setting `strictRTSJ` forces Jamaica VM to perform these checks.

If the option `strictRTSJ` is set, the following checks are performed and the corresponding exceptions are thrown:

`MemoryAccessError`: a `NoHeapRealtimeThread` attempts to access an object stored in normal Java heap, a `MemoryAccessError` is thrown.

`IllegalStateException`: a non-`RealtimeThread` attempts to enter a `javax.realtime.MemoryArea` or tries to access the scope stack by calling the methods `getCurrentMemoryArea`, `getMemoryAreaStackDepth`, `getOuterMemoryArea` or `getInitialMemoryAreaIndex`, which are defined in class `javax.realtime.RealtimeThread`.

Lazy linking is automatically disabled when `strictRTSJ` is set. This avoids runtime assignment errors due to lazily linked classes that may be allocated in a memory area that is incompatible with a current scoped memory allocation context when lazy linking is performed.

Option `-strictRTSJFromEnv=`*var*

The `strictRTSJFromEnv` option enables the application to read its setting of `strictRTSJ` from the specified environment variable. If this variable is not set, the value of the Boolean option `strictRTSJ` will be used. The value of the environment variable must be 0 (for `-strictRTSJ=false`) or 1 (for `-strictRTSJ=true`).

Option `-immortalMemorySize=`*n*`[K|M]`

The `immortalMemorySize` option sets the size of the immortal memory area, in bytes. The immortal memory can be accessed through the class `javax.realtime.ImmortalMemory`.

The immortal memory area is guaranteed never to be freed by the garbage collector. Objects allocated in this area will survive the whole application run.

Option `-immortalMemorySizeFromEnv=`*var*

The `immortalMemorySizeFromEnv` option enables the application to read its immortal memory size from the environment variable specified using this option. If this variable is not set, the immortal memory size specified using the option`-immortalMemorySize` will be used.

Option `-scopedMemorySize=`*n*`[K|M]`

The `scopedMemorySize` option sets the size of the memory that should be made available for scoped memory areas `javax.realtime.LTMemory` and `javax.realtime.VTMemory`. This memory lies outside of the normal Java heap, but it is nevertheless scanned by the garbage collector for references to the heap.

Objects allocated in scoped memory will never be reclaimed by the garbage collector. Instead, their memory will be freed when the last thread exits the scope.

Option `-scopedMemorySizeFromEnv=`*var*

The `scopedMemorySizeFromEnv` option enables the application to read its scoped memory size from the environment variable specified within. If this variable is not set, the scoped memory size specified using `-scopedMemorySize` *n* will be used.

Option `-physicalMemoryRanges[+]=`*range*{`,`*range*}

The `RawMemory` and `PhysicalMemory` classes in the `javax.realtime` package provide access to physical memory for Java applications. The memory ranges that may be accessed by the Java application can be specified using the option `physicalMemoryRanges`. The default behavior is that no access to physical memory is permitted by the application.

The `physicalMemoryRanges` option expects a list of address ranges. Each address range is separated by `..`, and gives the lower and upper address of the range: *lower*`..`*upper*. The lower address is inclusive and the upper address is

exclusive. I.e., the difference upper-lower gives the size of the accessible area. There can be an arbitrary number of memory ranges.

Example 1: `-physicalMemoryRanges=0x0c00..0x1000` will allow access to the memory range from address `0x0c00` to `0x1000`, i.e., to a range of 1024 bytes.

14.2.10 Native code

Native code is code written in a different programming language than Java (typically C or C++). This code can be called from within Java code using the Java Native Interface (JNI). Jamaica internally uses a more efficient interface, the Jamaica Binary Interface (JBI), for native calls into the VM and for compiled code.

Option `-object[+]=`*file*{`:`*file*}

The `object` option specifies object files that contain native code that has to be linked to the destination executable. Unlike other Java implementations, Jamaica does not access native code through shared libraries. Instead, the object files that contain native code referenced from within Java code are linked into the destination application file.

Multiple object files should be separated by the system specific path separator: colon "`:`" on Unix systems and semicolon "`;`" on Windows.

14.3 Builder Extended Usage

A number of extended options provide additional means for finer control of the Builder's operation for the more experienced user. The following sections list these extended options and describe their effect. Default values may be obtained by `jamaicabuilder -target=`*platform* `-xhelp`.

14.3.1 General

The following are general options which provide information about the Builder itself or enable the use of script files that specifying further options.

Option `-XdefineProperty[+]=`*name*`[=`*value*`]`

The `XdefineProperty` option sets a system property for the resulting binary. For security reasons, system properties set by the VM cannot by changed. The Unicode character U+EEEE is reserved and may not be used within the argument of the option.

Option -XdefinePropertyFromEnv[+]=*name*

At program start, the resulting binary will set a system property to the value of an environment variable. This feature can only be used if the target OS supports environment variables. For security reasons, system properties set by the VM cannot be changed.

Option -XignoreLineNumbers

Specifying the `XignoreLineNumbers` option instructs the Builder to remove the line number information from the classes that are built into the target application. The resulting information will have a smaller memory footprint and RAM demand. However, exception traces in the resulting application will not show line number information.

14.3.2 Classes, files and paths

These options allow to specify classes and paths to be used by the builder.

Option -XjamaicaHome=*path*

The `XjamaicaHome` option specifies the path to the Jamaica directory. The Jamaica home path can also be set with the environment variable `JAMAICA_HOME`.

Option -XjavaHome=*path*

The `XjamaicaHome` option specifies the path to the Jamaica directory. The Jamaica home path can also be set with the environment variable `JAMAICA_HOME`.

Option -Xbootclasspath[+]=*classpath*

The `Xbootclasspath` specifies path used for loading system classes.

Option -XlazyConstantStrings

Jamaica VM by default allocates all String constants at class loading time such that later accesses to these strings is very fast and efficient. However, this approach requires code to be executed for this initialization at system startup and it requires Java heap memory to store all constant Java strings, even those that are never touched by the application at run time

Setting option `-XlazyConstantStrings` causes the VM to allocate string constants lazily, i.e., not at class loading time but at time of first use of any constant string. This saves Java heap memory and startup time since constant strings that are never touched will not be created. However, this has the effect that accessing a constant Java string may cause an OutOfMemoryError.

Option `-XlazyConstantStringsFromEnv=`*var*

Causes the creation of an application that reads its `XlazyConstantStrings` setting from the specified environment variable. If this variable is not set, the value of boolean option `XlazyConstantStrings` will be used. The value of the environment variable must be `0` for `-XlazyConstantStrings=false` or `1` for `-XlazyConstantStrings=true`.

Option `-XnoMain`

The `XnoMain` option builds a standalone VM. Do not select a main class for the built application. Instead, the first argument of the argument list passed to the application will be interpreted as the main class.

Option `-XnoClasses`

The `XnoClasses` option does not include any classes in the built application. Setting this option is only needed when building the `jamaicavm` command itself.

14.3.3 Profiling and compilation

By default, the builder compiles all application classes and a predefined set of the system classes. Profiling and compilation options enable to fine tune the compilation process for optimal runtime performance of applications generated with the builder.

Option `-XprofileFilename=`*name*

The `XprofileFilename` option sets the name of the file to which profiling data will be written if profiling is enabled. If a profile filename is not specified then the profiling data will be written to a file named after the destination (see option `destination`) with the extension `.prof` added.

Option `-XprofileFilenameFromEnv=`*var*

The `XprofileFilenameFromEnv` creates an application that reads the name of a file for profiling data from the environment variable *var*. If this variable is not set, the name specified using `XprofileFilename` will be used (default: not used).

Option `-XfullStackTrace`

Compiled code usually does not contain full Java stack trace information if the stack trace is not required (as in a method with a try/catch clause or a synchronized method). For better debugging of the application, the `XfullStackTrace` option can be used to create a full stack trace for all compiled methods.

Option `-XexcludeLongerThan=`*n*

Compilation of large Java methods can cause large C routines in the intermediate code, especially when combined with aggressive inlining. Some C compilers have difficulties with the compilation of large routines. To enable the use of Jamaica with such C compilers, the compilation of large methods can be disabled using the option `XexcludeLongerThan`.

The argument specified to `XexcludeLongerThan` gives the minimum number of bytecode instructions a method must have to be excluded from compilation.

Option `-Xcc=`*cc*

The `Xcc` option specifies the C compiler to be used to compile intermediate C code that is generated by the Jamaica Builder.

Option `-XCFLAGS[+]=`*cflags*

The `XCFLAGS` option specifies the cflags for the invocation of the C compiler. Note that for optimizations the compiler independent option `-optimize` should be used.

Option `-Xld=`*linker*

The `Xld` option specifies the linker to be used to create a binary file from the object file generated by the C compiler.

Option `-XLDFLAGS[+]=`*ldflags*

The `XLDFLAGS` option specifies the ldflags for the invocation of the C linker.

Option `-dwarf2`

The `dwarf2` option generates a DWARF2 version of the application. DWARF2 symbols are needed for tracing Java methods in compiled code. Use this option with WCETA tools and binary debuggers.

Option `-XstripOptions=`*options*

The `XstripOptions` option specifies the strip options for the invocation of the stripper. See also option `Xstrip`.

Option `-Xlibraries[+]="`*library*{ *library*}`"`

The `Xlibraries` option specifies the libraries that must be linked to the destination binary. The libraries must include the option that is passed to the linker. Multiple libraries should be separated using spaces and enclosed in quotation marks. E.g., `-Xlibraries "m pthread"` causes linking against `libm` and `libpthread`.

Option `-XstaticLibraries[+]="`*library*{ *library*}`"`

The `XstaticLibraries` option specifies the libraries that must be statically linked to the destination binary. The libraries must include the option that is passed to the linker. Static linking creates larger executables, but may be necessary if the target system doesn't provide the library. Multiple libraries should be separated using spaces and enclosed in quotation marks.

Example: setting `-XstaticLibraries "m pthread"` causes static linking against `libm` and `libpthread`.

Option `-XlibraryPaths[+]=`*path*{`:`*path*}

The `XlibraryPaths` option adds the directories in the specified paths to the library search path. Multiple directories should be separated by the system specific path separator: colon "`:`" on Unix systems and semicolon "`;`" on Windows.

E.g., to use the directories `/usr/local/lib` and `/usr/lib` as library path, the option `-XlibraryPaths /usr/local/lib:/usr/lib` must be specified.

Option `-Xstrip=`*tool*

The `Xstrip` option uses the specified tool to remove debug information from the generated binary. This will reduce the size of the binary file by removing

information not needed at runtime.

Option -XnoRuntimeChecks

The `XnoRuntimeChecks` option disables runtime checks for compiled Java code. This option deactivates runtime checks to obtain better runtime performance. This may be used only for applications that do not cause any runtime checks to fail. Failure to run these checks can result in crashes, memory corruption and similar disasters. When untrusted code is executed, disabling these checks can cause vulnerability through attacks that exploit buffer overflows, type inconsistencies, etc.

The runtime checks disabled by this option are: checks for use of null pointers, out of bounds array indices, out of bounds string indices, array stores that are not compatible with the array element type, reference assignments between incompatible memory areas, division by zero and array instantiation with negative array size. These runtime checks usually result in throwing one of the following exceptions:

```
NullPointerException ArrayIndexOutOfBoundsException
StringIndexOutOfBoundsException ArrayStoreException
    IllegalAssignmentError ArithmeticException
            NegativeArraySizeException
```

When deactivated, the system will be in an undefined state if any of these conditions occurs.

Option -XavailableTargets

The `XavailableTargets` option lists all available target platforms of this Jamaica distribution.

14.3.4 Parallel Execution

The parallel version of JamaicaVM can execute several threads, including the garbage collection, in parallel and therefore improves the runtime performance when using multicore systems. Notice that you need to have an extra license to use the parallel version of JamaicaVM.

Option -Xcpus=*n1*{,*n2*} | *n1*..*n2* | all

Select the set of CPUs to use to run JamaicaVM on. The argument can be specified either as a set (e.g. `-Xcpus=0,1,2`) or a range (e.g. `-Xcpus=0..2`). All available CPUs are selected by using `-Xcpus=all`.

Option `-XcpusFromEnv=`*var*

The `XcpusFromEnv` option enables the application to read the set of CPUs to run on from the specified environment variable. If this variable is not set, the set specified using `-Xcpus` *set* will be used.

14.3.5 RTSJ settings

The following options set values that are relevant for the Real-Time Specification for Java extensions through classes javax.realtime.* that are provided by JamaicaVM.

Option `-XuseMonotonicClock`

On systems that provide a monotonic clock, setting this option enables use of this clock instead of the default realtime clock for RTSJ code.

Option `-XuseMonotonicClockFromEnv=`*var*

The `XuseMonotonicClockFromENv` option enables the application to read its setting of `XuseMonotonicClock` from the specified environment variable. If this variable is not set, the value of the option `XuseMonotonicClock` will be used. The environment variable must be set to 0 (`-XuseMonotonicClock=false`) or 1 (`-XuseMonotonicClock=true`).

14.3.6 Native code

Native code is code written in a different programming language than Java (typically C or C++). This code can be called from within Java code using the Java Native Interface (JNI). Jamaica internally uses a more efficient interface, the Jamaica Binary Interface (JBI), for native calls into the VM and for compiled code.

Option `-XenableDynamicJNILibraries`

The `XenableDynamicJNILibraries` option activates support for loading JNI libraries at runtime. This feature is currently implemented for the architectures x86 and ARM (ABI calling convention without FPU-support). On other systems JNI libraries must be linked at build time.

Option `-XloadJNIDynamic[+]=`*class*`|`*method*{ *class*`|` *method*}

The `XloadJNIDynamic` option will cause the Builder to know which native declared methods calls at runtime a dynamic library. Arguments have to be separated by spaces and enclosed in double quotes (`"`).

Here are examples of class arguments: `com.user.MyClass`, `com.user2.*` and `com...`

An example of a method argrument is

```
com.user.MyClass.toString()Ljava/lang/String.
```

Option `-Xinclude[+]=`*dirs*

The `Xinclude` option adds the specified directories to the include path. This path should contain the include files generated by `jamaicah` for the native code referenced from Java code. The include files are used to determine whether the Java Native Interface (JNI) or Jamaica Binary Interface (JBI) is used to access the native code.

This option expects a list of paths that are separated using the platform dependent path separator character (e.g., ':').

Option `-XobjectFormat=default | C | ELF | PECOFF`

The `XobjectFormat` option sets the object format to one of `default`, `C`, `PECOFF` and `ELF`.

Option `-XobjectProcessorFamily=`*type*

The `XobjectProcessorFamily` option sets the processor type for code generation. Available types are `none`, `i386`, `i486`, `i586`, `i686`, `ppc`, `sparc`, `arm`, `mips`, `sh`, `cris`, and `x86_64`. The processor type is only required if the ELF or PECOFF object formats are used. Otherwise the type may be set to `none`.

Option `-XobjectSymbolPrefix=`*prefix*

The `XobjectSymbolPrefix` sets the object symbol prefix, e.g., "_".

14.4 Environment Variables

The following environment variables control the `Builder`.

0	Normal termination
1	Error
2	Invalid argument
64	Insufficient memory
100	Internal error

Table 14.1: Jamaica Builder and jamaicah exitcodes

JAMAICA The Jamaica Home directory (*jamaica-home*). This variable sets the path of Jamaica to be used. Under Unix systems this must be a Unix style pathname, while under Windows this has to be a DOS style pathname.

JAMAICA_BUILDER_HEAPSIZE Initial heap size of the `Builder` program itself in bytes. Setting this to a larger value, e.g., "512M", will improve the `Builder` performance.

JAMAICA_BUILDER_MAXHEAPSIZE Maximum heap size of the `Builder` program itself in bytes. If the initial heap size of the `Builder` is not sufficient, it will increase its heap dynamically up to this value. To build large applications, you may have to set this maximum heap size to a larger value, e.g., "640M".

JAMAICA_BUILDER_JAVA_STACKSIZE Java stack size of the `Builder` program itself in bytes.

JAMAICA_BUILDER_NATIVE_STACKSIZE Native stack size of the `Builder` program itself in bytes.

14.5 Exitcodes

Tab. 14.1 lists the exit codes of the JamaicaVM Builder. If you get an exit code of an internal error please contact aicas support with a full description of the tool usage, command line options and input.

Chapter 15

The Jamaica ThreadMonitor

The JamaicaVM ThreadMonitor enables to monitor the realtime behavior of applications and helps developers to fine-tune the threaded Java applications running on Jamaica run-time systems. These run-time systems can be either the JamaicaVM or any application that was created using the Jamaica Builder.

The ThreadMonitor tool collects and presents data sent by the scheduler in the Jamaica run-time system, and is invoked with the `jamaica_threadmonitor` command. When ThreadMonitor is started, it presents the user a control window (see Fig. 15.1).

15.1 Run-time system configuration

The event collection for ThreadMonitor in the Jamaica run-time system is controlled by two system properties:

- `jamaica.scheduler_events_port`
- `jamaica.scheduler_events_port_blocking`

To enable the event collection in the JamaicaVM, a user sets the value of one of these properties to the port number to which the ThreadMonitor GUI will connect later. If the user chooses the `blocking` property, the VM will stop after the bootstrapping and before the main method is invoked. This enables a developer to investigate the startup behavior of an application.

```
> jamaicavm -cp classes -Djamaica.scheduler_events_port=2712 \
> HelloWorld
**** accepting Scheduler Events Recording requests on port #2712
              Hello        World!
           Hello       World!
        Hello     World!
```

```
    Hello    World!
  Hello   World!
Hello  World!
[...]
```

When event collection is enabled, the requested events are written into a buffer and sent to the ThreadMonitor tool by a high priority periodic thread. The amount of buffering and the time periods can be controlled from the GUI.

15.2 Control Window

The ThreadMonitor control window is the main interface to control recording scheduler data from applications running with Jamaica.

On the right hand side of the window, IP address and port of the VM to be monitored may be entered.

The following list gives a short overview on which events data is collected:

- Thread state changes record how the state of a thread changes over time including which threads cause state changes in other threads.
- Thread priority changes show how the priority changed due to explicit calls to `Thread.setPriority()` as well as adjustments due to priority inheritance on Java monitors.
- Thread names show the Java name of a thread.
- Monitor enter/exit events show whenever a thread enters or exits a monitor successfully as well as when it blocks due to contention on a monitor.
- GC activity records when the incremental garbage collector does garbage collection work.
- Start execution shows when a thread actually starts executing code after it was set to be running.
- Reschedule shows the point when a thread changes from running to ready due to a reschedule request.
- All threads that have the state ready within the JamaicaVM are also ready to run from the OS point of view. So it might happen that the OS chooses a thread to run that does not correspond with the running thread within the VM. In such cases, the thread chosen by the OS performs a yield to allow a different thread to run.

Name	Value
Event classes	Selection of event classes that the run-time system should send.
IP Address	The IP address of the run-time system.
Port	The Port where the runtime system should be contacted (see § 15.1).
Buffer Size	The amount of memory that is allocated within the run-time system to store event data during a period.
Sample Period	The period length between sending data.
Start Recording	When pressed connects the ThreadMonitor to the run-time systems and collects data until pressed again.

Table 15.1: ThreadMonitor Controls

- User events contain user defined messages and can be triggered from Java code. To trigger a user event, the following method can be used:

  ```
  com.aicas.jamaica.lang.Scheduler.recordUserEvent
  ```

  For its signature, please consult the API doc of the `Scheduler` class.

- Allocated memory gives an indication of the amount of memory that is currently allocated by the application. The display is relatively coarse, changes are only displayed if the amount of allocated memory changes by 64kB. A vertical line gives indicates what thread performed the memory allocation or GC work that caused a change in the amount of allocated memory.

When ThreadMonitor is started it presents the user a control window Fig. 15.1.

15.2.1 Control Window Menu

The control window's menu permits only three actions:

15.2.1.1 File/Open...

This menu item will open a file requester to load previously recorded scheduler data that was saved through the data window's "File/Save as..." menu item, see § 15.3.2.2.

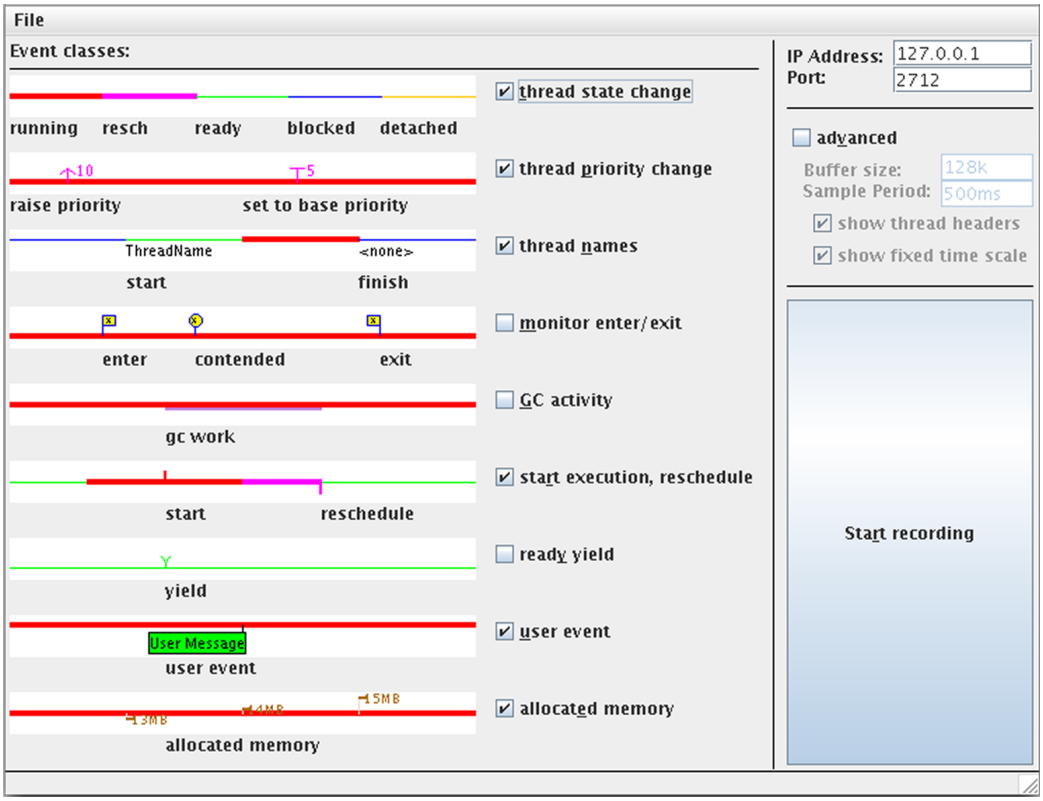


Figure 15.1: Control view of the ThreadMonitor

15.2.1.2 File/Close

Select this menu item will close the control window, but it will leave all other windows open.

15.2.1.3 File/Quit

Select this menu item will close all windows of the ThreadMonitor tool and quit the application.

15.3 Data Window

The data window will display scheduler data that was recorded through "Start/Stop recording" in the control window or that was loaded from a file.

To better understand the ThreadMonitor output, it is helpful to have some understanding of the JamaicaVM scheduler. The JamaicaVM scheduler provides real-time priority enforcement within Java programs on operating systems that do not offer strict priority based scheduling (e.g. Linux for user programs). The scheduler reduces the overhead for JNI calls and helps the operating system to better schedule CPU resources for threads associated with the VM. These improvements let the JamaicaVM integrate better with the target OS and increase the throughput of threaded Java applications.

The VM scheduler controls which thread runs within the VM at any given time. This means it effectively protects the VM internal data structures like the heap from concurrent modifications. The VM scheduler does not replace, but rather supports, the operating system scheduler. This allows, for example, for a light implementation of Java monitors instead of using heavy system semaphores.

All threads created in the VM are per default attached to the VM (i.e. they are controlled by the VM scheduler). Threads that execute system calls must detach themselves from the VM. This allows the VM scheduler to select a different thread to be the running thread within the VM while the first thread for example blocks on an IO request. Since it is critical that no thread ever blocks in a system call while it is attached, all JNI code in the JamaicaVM is executed in detached mode.

For the interpretation of the ThreadMonitor data, the distinction between attached and detached mode is important. A thread that is detached could still be using the CPU, meaning that the thread that is shown as running within the VM might not actually be executing any code. Threads attached to the VM may be in the states running, rescheduling, ready, or blocked. Running means the thread that currently executes within the context of the VM. Rescheduling is a sub state of the running thread. The running thread state is changed to rescheduling when another thread becomes more eligible to execute. This happens when a thread of

higher priority becomes ready either by unblocking or attaching to the VM. The running thread will then run to the next synchronization point and yield the CPU to the more eligible thread. Ready threads are attached threads which can execute as soon as no other thread is more eligible to run. Attached threads may block for a number of reasons, the most common of which are calls to Thread.sleep, Object.wait, and entering of a contended monitor.

15.3.1 Data Window Navigation

The data window permits easy navigation through the displayed scheduler data. Two main properties can be changed: The time resolution can be contracted or expanded, and the total display can be enlarged or reduced (zoom in and zoom out). Four buttons on the top of the window serve to change these properties.

15.3.1.1 Selection of displayed area

The displayed area can be selected using the scroll bars or via dragging the contents of the window while holding the left mouse button.

15.3.1.2 Time resolution

The displayed time resolution can be changed via the buttons “expand time” and “contract time” or via holding down the left mouse button for expansion or the middle mouse button for contraction. Instead of the middle mouse button, the control key plus the left mouse button can also be used.

15.3.1.3 Zoom factor

The size of the display can be changed via the buttons “zoom in” and “zoom out” or via holding down shift in conjunction with the left mouse button for enlargement or in conjunction with the right mouse button for shrinking. Instead of shift and the middle mouse button, the shift and the control key plus the left mouse button can also be used.

15.3.2 Data Window Menu

The data window’s menu offers the following actions.

15.3.2.1 File/Open...

This menu item will open a file requester to load previously recorded scheduler data that was saved through the data window's "File/Save as..." menu item, see § 15.3.2.2.

15.3.2.2 File/Save as...

This menu item permits saving the displayed scheduler data, such that it can later be loaded through the control window's "File/Open..." menu item, see § 15.2.1.1.

15.3.2.3 File/Close

Select this menu item will close the data window, but it will leave all other windows open.

15.3.2.4 File/Quit

Select this menu item will close all windows of the ThreadMonitor tool and quit the application.

15.3.2.5 Options/Grid

Selecting this option will display light gray vertical grid lines that facilitate relating a displayed event to the point on the time scale.

15.3.2.6 Options/Thread Headers

If this option is selected, the left part of the window will be used for a fixed list of thread names that does not participate in horizontal scrolling.

15.3.2.7 Options/Thread Headers

If this option is selected, the top part of the window will be used for a fixed time scale that does not participate in vertical scrolling. This is useful in case many threads are displayed and the time scale should remain visible when scrolling through these threads.

15.3.2.8 Navigate/Fit Width

This menu item will change the time contraction such that the whole data fits into the current width of the window.

15.3.2.9 Navigate/Fit Height

This menu item will change the zoom factor such that the whole data fits into the current height of the window.

15.3.2.10 Navigate/Fit Window

This menu item will change the time contraction and the zoom factor such that the whole data fits into the current size of the data window.

15.3.2.11 Tools/Worst-Case Execution Times

This menu item will start the execution time analysis and show the Worst-Case Execution Time window, see § 15.3.5.

15.3.2.12 Tools/Reset Monitors

The display of monitor enter and exit events can be suppressed for selected monitors via a context menu on an event of the monitor in questions. This menu item re-enables the display of all monitors.

15.3.3 Data Window Context Window

The data window has a context menu that appears when pressing the right mouse button over a monitor event. This context window permits to suppress the display of events related to a monitor. This display can be re-enabled via the Tools/Reset Monitors menu item.

15.3.4 Data Window Tool Tips

When pointing onto a thread in the data window, a tool tip appears that display information on the current state of this thread including its name, the state (running, ready, etc.) and the thread's current priority.

15.3.5 Worst-Case Execution Time Window

Through this window, the ThreadMonitor tool enables the determination of the maximum execution time that was encountered for each thread within recorded scheduler data. If the corresponding menu item was selected in the data window (see § 15.3.2.11), execution time analysis will be performed on the recorded data and this window will be displayed.

The window shows a table with one row per thread and the following data given in each column.

Thread # gives the Jamaica internal number of this thread. Threads are numbered starting at 1. One Thread number can correspond to several Java threads in case the lifetime of these threads does not overlap.

Thread Name will present the Java thread name of this thread. In case several threads used the same thread id, this will display all names of these threads separated by vertical lines.

Worst-case execution time presents the maximum execution time that was encountered in the scheduler data for this thread. This column will display "N/A" in case no releases where found for this thread. See below for a definition of execution time.

Occurred at gives the point in time within the recording at which the release that required the maximum execution time started. A mouse click on this cell will cause this position to be displayed in the center of the data window the worst-case execution time window was created from. This column will display "N/A" in case no Worst-case execution time was displayed for this thread.

 # Releases is the number of releases that of the given thread that where found during the recording. See below for a definition of a release.

Average time is the average execution time for one release of this thread. See below for a definition of execution time.

Comment will display important additional information that was found during the analysis. E.g., in case the data the analysis is based on contains overflows, i.e. periods without recorded information, these times cannot be covered by this analysis and this will be displayed here.

15.3.5.1 Definitions

Release of a thread T is a point in time at which a waiting thread T becomes ready to run that is followed by a point in time at which it will block again waiting for the next release. I.e., a release contains the time a thread remains ready until it becomes running to execute its job, and it includes all the time the thread is preempted by other threads or by activities outside of the VM.

Execution Time of a release is the time that has passed between a release and the point at which the thread blocked again to wait for the next release.

15.3.5.2 Limitations

The worst-case execution times displayed in the worst-case execution times window are based on the measured scheduling data. Consequently, they can only display the worst-case times that were encountered during the actual run, which may be fully unrelated to the theoretical worst-case execution time of a given thread. In addition to this fundamental limitation, please be aware of the following detailed limitations:

Releases are the points in time when a waiting thread becomes ready. If a release is caused by another thread (e.g., via Java function `Object.notify()`), this state change is immediate. However, if a release is caused by a timeout of a call to `Object.wait()`, `Thread.sleep()`, `RealtimeThread.waitForNextPeriod()` or similar functions, the state change to ready may be delayed if higher priority threads are running and the OS does not assign CPU time to the waiting thread. A means to avoid this inaccuracy is to use a high-priority timer (e.g., class `javax.realtime.Timer`) to wait for a release.

Blocking waits within a release will result in the worst-case execution time analysis to treat one release as two independent releases. Therefore, the analysis is wrong for tasks that perform blocking waits during a release. Any blocking within native code, e.g., blocking I/O operations, is not affected by this, so the analysis can be used to determine the execution times of I/O operations.

Chapter 16

Jamaica and the Java Native Interface (JNI)

The Java Native Interface (JNI) is a standard mechanism for interoperability between Java and native code, i.e., code written with other programming languages like C. Jamaica implements version 1.4 of the Java Native Interface. Creating and destroying the vm via the Invocation API is currently not supported.

16.1 Using JNI

Native code that is interfaced through the JNI interface is typically stored in shared libraries that are dynamically loaded by the virtual machine when the application uses native code. Jamaica supports this on many platforms, but since dynamically loaded libraries are usually not available on small embedded systems that do not provide a file system, Jamaica also offers a different approach. Instead of loading a library at runtime, you can statically include the native code into the application itself, i.e., link the native object code directly with the application.

The Builder allows direct linking of native object code with the created application through the option `-object` *file* or `-XstaticLibraries` *file*. Multiple files can be linked. Separate the file names with spaces and enclose the whole option argument within "`"`" (double quotes). All object files containing native code should be presented to the Builder using this option.

Building an application using native code on a target requiring manual linking may require providing these object files to the linker. Here is a short example on the use of the Java Native Interface with Jamaica. This example simply writes a value to a hardware register using a native method. We use the file `JNITest.java`, which contains the following code:

```
public class JNITest {
  static native int write_HW_Register(int address,
```

```
                                        int value);

  public static void main(String args[]) {
    int value;

    value = write_HW_Register(0xfc000008,0x10060);
    System.out.println("Result: "+value);
  }
}
```

Jamaica provides a tool, `jamaicah`, for generating C header files that contain the function prototypes for all native methods in a given class. Note that `jamaicah` operates on Java class files, so the class files have to be created first using `jamaicac` as described in § 12. The header file for `JNITest.java` is created by the following sequence of commands:

```
> jamaicac JNITest.java
> jamaicah JNITest
Reading configuration from '/usr/local/jamaica/etc/jamaicah.conf'...
+ JNITest.h (header)
```

This created the include file `JNITest.h`:

```
/* DO NOT EDIT THIS FILE - it is machine generated */
#include <jni.h>
/* Header for class JNITest */

#ifndef _Included_JNITest
#define _Included_JNITest
#ifdef __cplusplus
extern "C" {
#endif
/* Class:     JNITest
 * Method:    write_HW_Register
 * Signature: (II)I */
#ifdef __cplusplus
  extern "C"
#endif
JNIEXPORT jint JNICALL Java_JNITest_write_1HW_1Register(JNIEnv *env,
                                                        jclass c,
                                                        jint v0,
                                                        jint v1);


#ifdef __cplusplus
}
#endif
#endif
```

The native code is implemented in `JNITest.c`.

```
#include "jni.h"
#include "JNITest.h"
#include <stdio.h>

JNIEXPORT jint JNICALL
Java_JNITest_write_1HW_1Register(JNIEnv *env,
                                 jclass c,
                                 jint   v0,
                                 jint   v1)
{
  printf("Now we could write the value %i into "
    "memory address %x\n", v1, v0);
  return v1; /* return the "written" value */
}
```

Note that the mangling of the Java name into a name for the C routine is defined in the JNI specification. In order to avoid typing errors, just copy the function declarations from the generated header file.

A C compiler is used to generate an object file. Here, `gcc` — the GNU C compiler — is used, but other C compilers should also work. Note that the include search directories provided with the option `-I` may be different on your system.

For Unix users using `gcc` the command line is:

```
> gcc -I/usr/local/jamaica/target/linux-x86/include \
>   -c -m32 JNITest.c
```

For Windows user using the Visual Studio C compiler the command line is:

```
> cl /Ic:\Programs\jamaica\target\windows-x86\include
    /c JNITest.c
```

(This should be entered as a single line.) Finally, the Builder can be called to generate a binary file which contains all necessary classes as well as the object file with the native code from `JNITest.c`. The Builder is called by:

```
> jamaicabuilder -object=JNITest.o JNITest
Reading configuration from
'usr/local/jamaica-6.2/target/linux-x86/etc/jamaica.conf'...
Jamaica Builder Tool 6.2 Release 0 (build 7608)
(User: EVALUATION USER, Expires: 2012.11.04)
Generating code for target 'linux-x86', optimization 'speed'
 + tmp/PKG__V5df018cb46f7a904__.c
 [...]
 + tmp/JNITest__.c
 + tmp/JNITest__.h
 * C compiling 'tmp/JNITest__.c'
 [...]
 + tmp/JNITest__nc.o
```

```
 * linking
 * stripping
Application memory demand will be as follows:
                       initial               max
Thread C    stacks:    1024KB (=   8* 128KB)   63MB (= 511* 128KB)
Thread Java stacks:     128KB (=   8*  16KB) 8176KB (= 511*  16KB)
Heap Size:             2048KB                256MB
GC data:                128KB                 16MB
TOTAL:                 3328KB                343MB
```

The created application can be executed just like any other executable:

```
> ./JNITest
Result: 65632
Now we could write the value 65632 into memory address fc000008
```

16.2 The Jamaicah Command

A variety of arguments control the work of the jamaicah tool. The command line syntax is as follows:

`jamaicah [`*options*`]` *class*

The class argument identifies the class for which native headers are generated.

16.2.1 General

These are general options providing information about jamaicah itself.

Option `-help (--help, -h, -?)`

Display the usage of the jamaicah tool and a short description of all possible standard command line options.

Option `-Xhelp`

Display the usage of the jamaicah tool and a short description of all possible standard and extended command line options. Extended command line options are not needed for normal control of the jamaicah command. They are used to configure tools and options and to provide tools required internally for Jamaica VM development.

Option `-jni`

Create Java Native Interface header files for the native declarations in the provided Java class files. This option is the default and hence does not need to be specified explicitly.

Option `-d=`*directory*

Specify output directory for created header files. The filenames are deduced from the full qualified Java class names where "." are replaced by "_" and the extension "`.h`" is appended.

Option `-o=`*file*

Specify the name of the created header file. If not set the filename is deduced from the full qualified Java class name where "." are replaced by "_" and the extension "`.h`" is appended.

Option `-includeFilename=`*file*

Specify the name of the include file to be included in stubs.

Option `-version`

Prints the version of the jamaicah tool and exits.

16.2.2 Classes, files, and paths

Option `-classpath (-cp)[+]=`*classpath*

Specifies default path used for loading classes.

Option `-bootclasspath (-Xbootclasspath)[+]=`*classpath*

Specifies default path used for loading system classes.

Option `-classname[+]="`*class*{ *class*}`"`

Generate header files for the listed classes. Multiple items must be separated by spaces and enclosed in double quotes.

16.2.3 Environment Variables

The following environment variables control `jamaicah`.

`JAMAICA_JAMAICAH_HEAPSIZE` Initial heap size of the `jamaicah` program itself in bytes.

`JAMAICA_JAMAICAH_MAXHEAPSIZE` Maximum heap size of the `jamaicah` program itself in bytes. If the initial heap size of `jamaicah` is not sufficient, it will increase its heap dynamically up to this value.

Chapter 17

Building with Apache Ant

Apache Ant is a popular build tool in the Java world. Ant *tasks* for the Jamaica Builder and other tools are available. In this chapter, their use is explained.

Ant build files (normally named `build.xml`) are created and maintained by the Jamaica Eclipse Plug-In (see § 4). They may also be created manually. To obtain Apache Ant, and for an introduction, see the web page `http://ant.apache.org`. Apache Ant is not provided with Jamaica. In the following sections, basic knowledge of Ant is presumed.

17.1 Task Declaration

Ant tasks for the Jamaica Builder, `jamaicah` and a task for calling the C compiler are provided. The latter two are useful for building programs that include native code via JNI with Ant. In order to use these tasks, `taskdef` directives are required. The following code should be placed after the opening `project` tag of the build file:

```
<taskdef name="jamaicabuilder"
  classpath="jamaica-home/lib/JamaicaTools.jar"
  classname="com.aicas.jamaica.tools.ant.JamaicaTask" />
<taskdef name="jamaicacc"
  classpath="jamaica-home/lib/JamaicaTools.jar"
  classname="com.aicas.jamaica.tools.ant.JamaicaCCTask" />
<taskdef name="jamaicah"
  classpath="jamaica-home/lib/JamaicaTools.jar"
  classname="com.aicas.jamaica.tools.ant.JamaicahTask" />
```

The task names are used within the build file to reference these tasks. They may be chosen arbitrarily for stand-alone build files. For compatibility with the Eclipse Plug-In, the names `jamaicabuilder` and `jamaicah` should be used.

17.2 Task Usage

All Jamaica Ant tasks obtain the root directory of the Jamaica installation from the environment variable `JAMAICA`. Alternatively, the attribute `jamaica` may be set to *jamaica-home*.

17.2.1 Jamaica Builder and Jamaicah

Tool options are specified as nested elements. For each tool option *−option*, a corresponding *option* element is available in the Ant task of that tool. These option elements accept the attributes shown in the following table. All attributes are optional.

Attribute	Description	Required
`value`	Option argument	For options that require an argument.
`enabled`	Whether the option is passed to the tool.	No (default `true`)
`append`	Value is appended to the value stored in the tool's configuration file (+= syntax).	No (default `false`)

Although Ant buildfiles are case-insensitive, the precise spelling of the option name should be preserved for compatibility with the Eclipse Plug-In.

The following example shows an Ant target for executing the Jamaica Builder.

```
<target name="build_app">
  <jamaicabuilder jamaica="/usr/local/jamaica">
    <target          value="linux-x86"/>
    <classpath       value="classes"/>
    <classpath       value="extLib.jar"/>
    <interpret       value="true" enabled="false"/>
    <heapSize        value="32M"/>
    <Xlibraries      value="extLibs" append="true"/>
    <XdefineProperty value="window.size=800x600">
    <main            value="Application"/>
  </jamaicabuilder>
</target>
```

This is equivalent to the following command line:

```
/usr/local/jamaica/bin/jamaicabuilder
  -target=linux-x86
  -classpath=classes:extLib.jar
  -heapSize=32M
  -Xlibraries+=extLibs
  -XdefineProperty=window.size=800x600
  Application
```

Note that some options take arguments that contain the equals sign. For example, the argument to `XdefineProperty` is of the form *property=value*. As shown in the example, the entire argument should be placed in the `value` attribute literally. Ant pattern sets and related container structures are currently not supported by the Jamaica Ant tasks.

17.2.2 C Compiler

The C Compiler task provides an interface to the target-specific compiler, which is called by the Builder.

Attribute	Description	Required
`configuration`	Jamaica configuration file from which default settings are taken.	No (defaults to the Jamaica configuration file of the target platform given via the `target` attribute.)
`target`	Platform for which to compile.	No (default: host platform)
`source`	C source file	Yes
`output`	Output file	Yes
`defines`	Comma separated list of macros. These are set to the compiler's default (usually 1). Providing definitions for macros is not supported.	No (default: setting from the configuration file)
`includepath`	Search path for header files.	No (default: setting from the configuration file)
`shared`	If set, add compiler flags needed for building shared libraries.	No (default `false`)
`compilerflags`	Space separated list of command line arguments passed to the compiler verbatim. This extends the default setting.	No (default: setting from the configuration file)
`verbose`	If set, print the generated C Compiler command line.	No (default `false`)
`debug`	Generate code with asserts enabled.	No (default `false`)
`gprof`	Generate code for GNU gprof. Not supported on all platforms.	No (default: setting from the configuration file)

Attribute	Description	Required
dwarf2	Generate debug information compatible with DWARF version 2. Not supported on all platforms.	No (default: setting from the configuration file)

The nested element <includepath> extends the include path set via attribute or from the configuration file. It is a path-like structure; for more information see the respective chapter in the Ant Manual [1]. The nested element is useful for extending the default include path from the configuration file.

This task is used in the test_jni example, which may be consulted for an illustration.

17.3 Setting Environment Variables

The Jamaica Ant tasks do support two additional nested elements, <env> and <envpropertyset>, that can be used to provide environment variables to the tool. This is normally only required if the target-specific configuration requires certain environment variables to be set.

For example, when building for VxWorks 6.6, it may be necessary to provide environment variables in the following way:

```
<jamaicabuilder jamaica="/usr/local/jamaica">
  <env key="WIND_HOME" value="/opt/WindRiver"/>
  <env key="WIND_BASE" value="/opt/WindRiver/vxworks-6.6"/>
  <env key="WIND_USR"  value="/opt/WindRiver/target/usr"/>
  ...
</jamaicabuilder>
```

or alternatively, using a PropertySet:

```
<property name="WIND_HOME" value="/opt/WindRiver"/>
<property name="WIND_BASE" value="/opt/WindRiver/vxworks-6.6"/>
<property name="WIND_USR"  value="/opt/WindRiver/target/usr"/>

<jamaicabuilder jamaica="/usr/local/jamaica">
  <envpropertyset>
    <propertyref prefix="WIND_"/>
  </envpropertyset>
  ...
</jamaicabuilder>
```

For more information about the usage of these two elements, please refer to their respective chapters in the Ant Manual [1].

Part IV

Additional Information

Appendix A

FAQ — Frequently Asked Questions

Check here first when problems occur using JamaicaVM and its tools.

A.1 Software Development Environments

Q: I use Eclipse to develop my Java applications. Is there a plug-in available which will help me to use JamaicaVM and the Builder from within Eclipse?

A: Yes. There is a plugin available that will help you to configure the Builder download and execute your application on your target. For more information, see `http://www.aicas.com/eclipse.html`. For a quick start, you can use the Eclipse Update Site Manager with the following Update Site URL: `http://aicas.com/download/eclipse-plugin`. This conveniently downloads and installs the plugin.

Q: When I set up a Java Runtime Environment (JRE) with the JamaicaVM Eclipse Plugin, the bootclasses (`rt.jar`) are set up to be taken from the host platform. Is this safe when developing for the target platform?

A: The `rt.jar` configured in the runtime environment will be used by Eclipse for generating Java Bytecode and for running the Jamaica host VM. Code for the target platform is generated by the JamaicaVM Builder, which automatically chooses the correct `rt.jar`. Since the Java APIs defined by the host and target `rt.jar` are compatible (except if the target is a profile other than the Java Standard Edition), the Java Bytecode generated by Eclipse will be compatible regardless of whether the `rt.jar` is for the host or the target, and it is sufficient that the Builder chooses the correct `rt.jar`.

A.2 JamaicaVM and Its Tools

A.2.1 JamaicaVM

Q: When I try to execute an application with the JamaicaVM I get the error message `OUT OF MEMORY`. What can I do?

A: The JamaicaVM has a predefined setting for the internal heap size. If it is exhausted the error message `OUT OF MEMORY` is printed and JamaicaVM exits with an error code. The predefined heap size of 256MB is usually large enough, but for some applications it may not be sufficient. You can set the heap size via the `jamaicavm` options `Xmx`*size*, via the environment variable `JAMAICAVM_MAXHEAPSIZE`, e.g., under `bash` with

```
export JAMAICAVM_MAXHEAPSIZE=268435456
```

or, when using the builder, via the builder option `maxHeapSize`.

Q: When the built application terminates I see some output like `WARNING: termination of thread 7 failed`. What is wrong?

A: At termination of the application the JamaicaVM tries to shutdown all running threads by sending some signal. If a thread is stuck in a native function, e.g., waiting in some OS kernel call, the signal is not received by the thread and there is no response. In that case the JamaicaVM does a hard-kill of the thread and outputs the warning. Generally, the warning can simply be ignored, but be aware that a hard-kill may leave the OS in an unstable state, or that some resources (e.g., memory allocated in a native function) can be lost. Such hard-kills can be avoided by making sure no thread gets stuck in a native-function call for a long time (e.g., more than 100ms).

Q: At startup JamaicaVM prints this warning:

```
CPU rate unknown, please set property >>jamaica.cpu_mhz<<.
Measured rate: 1799.6MHz
```

Why could this be a problem?

A: The CPU cycle counter is used on some systems to measure time by JamaicaVM. In particular, this is used by cost monitoring within the RTSJ and by code that uses the class `com.aicas.jamaica.lang.CpuTime`. To map the number of CPU cycles to a time measured in seconds (or nanoseconds), the CPU frequency is required. For most target systems, JamaicaVM

does not have a means of determining the CPU frequency. Instead, it will fall back to measure the frequency and print this warning.

Since the measurement has a relevant runtime overhead and brings some inaccuracy, it is better to specify the frequency via setting the Java property `jamaica.cpu_mhz` to the proper value. Care is needed since setting the property to an incorrect value will result in cost enforcement to be too strict (if set too low) or too lax (if set too high).

A.2.2 JamaicaVM Builder

Q: When I try to start a Jamaica compiled executable in a Linux 2.4.x environment, I get an error message like the following:

```
__alloc_pages: 0-order allocation failed (gfp=0x1d0/0)
 from c0123886
```

A: This is a bug in the Linux 2.4.10 kernel. Please use a newer kernel version. The problem seems to occur if the amount of allocated memory is close to the amount of available memory, which is usually no problem if you use Jamaica on a desktop PC, but occurs quite often on embedded systems running Linux 2.4.10 and Jamaica (e.g., the DILNet/PC).

Q: When I try to compile an application with the Builder I get the error message `OUT OF MEMORY`. What can I do?

A: The Builder has a predefined setting for the internal heap size. If the memory space is exhausted, the error message `OUT OF MEMORY` is printed and Builder exits with an error code. The predefined maximum heap size (1024MB) is usually large enough, but for some applications it may not be sufficient. You can set the maximum heap size via the environment variable `JAMAICA_BUILDER_MAXHEAPSIZE`, e.g., under `bash` with the following command:

```
> export JAMAICA_BUILDER_MAXHEAPSIZE=1536MB
```

Q: When I try to compile an application with the Builder I get the error message:

```
jamaicabuilder: I/O error while executing C-compiler:
Executing 'gcc' failed: Cannot allocate memory.
```

A: There is not enough memory available to compile the C files generated by the Builder. You can increase the available memory on your system or reduce the predefined heap size of the Builder, e.g. under `bash` with the following command:

```
> export JAMAICA_BUILDER_HEAPSIZE=150MB
> export JAMAICA_BUILDER_MAXHEAPSIZE=300MB
```

Be aware that you could get an `OUT OF MEMORY` error if the heap size is too small to build your application.

Q: When I build an application which contains native code it seems that some fields of the class files can be accessed with the function `GetFieldID()` from the native code, but some others not. What happened to those fields?

A: If an application is built, the Builder removes from classes all unreferenced methods and fields. If a field in a class is only referenced from native code the Builder can not detect this reference and protect the field from the smart-linking-process. To avoid this use the `includeClasses` option with the class containing the field. This will instruct the Builder to fully include the specified class(es).

Q: When I build an application with the Builder I get some warning like the following:

```
WARNING: Unknown native interface type of class 'name'
 (name.h) - assume JNI calling convention
```

Is there something wrong?

A: In general, this is not an error. The Builder outputs this warning when it is not able to detect whether a native function is implemented using JNI (the standard Java native interface; see chapter § 16) or JBI (a Jamaica specific, more efficient native interface used by the $Jamaica; boot classes). Usually this means the appropriate header file generated with some prototype tool like `jamaicah` is not found or not in the proper format. To avoid this warning, recreate the header file with `jamaicah` and place it into a directory that is passed via the builder argument `Xinclude`.

Q: How can I set properties (using `-D`*name*=*value*) for an application that was built using the builder?

A: For commands like `jamaicavm`, parsing of arguments like `-D`*name*=*value* stops at the name of the main class of the application. After the application has been built, the main class is an implicit argument, so there is no direct way to provide additional options to the VM. However, there is a way out of this problem: the Builder option `-XnoMain` removes the implicit argument for the main class, so `jamaicavm`'s normal argument parsing is used

to find the main class. When launching this application, the name of the main class must then be specified as an argument, so it is possible to add additional VM options such as `-D`*name*=*value* before this argument.

Q: When I run the Builder an error "`exec fail`" is reported when the intermediate C code should be compiled. The exit code is 69. What happened?

A: An external C compiler is called to compile the intermediate C code. The compiler command and arguments are defined in `etc/jamaica.conf`. If the compiler command can not be executed the Builder terminates with an error message and the exit code 69 (see list of exit codes in the appendix). Try to use the verbose output with the option `-verbose` and check if the printed compiler command call can be executed in your command shell. If not check the parameters for the compiler in `etc/jamaica.conf` and the PATH environment variable.

Q: Can I build my own VM as an application which expects the name of the main class on the command line like JamaicaVM does?

A: With the Builder a standalone VM can be built with the option `-XnoMain`. If this option is specified, the Builder does not expect a main class while compiling. Instead, the built application expects the main class later after startup on the command line. Some classes or resources can be included in the created VM, e.g., a VM can be built including all classes of the selected API except the main program with main class.

A.2.3 Third Party Tools

Q: I would like to use JamaicaVM on Windows. Do I need Microsoft Visual Studio?

A: Visual Studio is only required when developing for Windows or Windows CE. If developing for other operating systems, the tool and SDK locations (see § 2.1) may be left empty.

A.3 Supported Technologies

A.3.1 Cryptography

Q: Cryptographic protocols such as `https` do not work.

A: Due to export restrictions, cryptographic support provided by Jamaica is limited. For JamaicaVM 6.2 Release 4 and earlier the limit defaults to 48 Bit. For JamaicaVM 6.2 Release 5 and later, the limit was increased to support default configurations of common protocols such as `https`. In order to increase the cryptographic strength of Jamaica beyond the default, jurisdiction policy files that permit strong encryption must added to the the Jamaica installation.

In order to obtain suitable policy files for your needs, aicas may be contacted at one of the addresses given in the front matter of this manual. Jurisdiction policy files will be provided in accordance to export regulations. Currently, a stronger version of `local_policy.jar` is sufficient, and either `limited_local_policy.jar` or `unlimited_local_policy.jar` will be provided.

Different policies may be installed simultaneously by copying different policy files to the `lib/security` subfolder of the home folder of the Java runtime system for the desired platform:

jamaica-home`/target/`*platform*`/lib/security`

The builder option `-setLocalCryptoPolicy` may be used to choose the policy file to be included under the name `local_policy.jar` into an application built with Jamaica.

To use stronger encryption with the `jamaicavm` command, i.e., without using the Jamaica builder, you have to replace the `local_policy.jar` file by `limited_local_policy.jar` or `unlimited_local_policy.jar`. Alternatively, the `java.home` property may be set to another folder containing the folder `lib/security`, and in which the desired policy file was put and renamed to `local_policy.jar`.

Before replacing policy files, it is recommended to rename the existing files, so the original settings can be restored easily.

A.3.2 Fonts

Q: How can I change the mapping from Java fonts to native fonts?

A: The mapping between Java font names and native fonts is defined in the `font.properties` file. Each target system provides this file with useful default values. An application developer can provide a specialized version for this file. To do this the new mapping file must exist in the classpath at build time. The file must be added as a resource to the final application

by adding `-resource+=`*path* where *path* is a path relative to a classpath root. Setting the system property "jamaica.fontproperties" with the option `-XdefineProperty=jamaica.fontproperties=`*path* will provide the graphics environment with the location of the mapping file.

Q: Why do fonts appear different on host and target?

A: Jamaica relies on the target graphics system to render true type fonts. Since that renderer is generally a different one than on the host system it is possible that the same font is rendered differently.

A.3.3 Serial Port

Q: How can I access the serial port (UART) with Jamaica?

A: You can use RXTX which is available for Linux, Windows, Solaris and as source code at `http://users.frii.com/jarvi/rxtx`. Get further information there.

Q: Can I use the Java Communications API?

A: The Java Communications API (also known as `javax.comm`) is not supported by Jamaica. Existing applications can be ported to RXTX easily.

A.3.4 Realtime Support and the RTSJ

Q: Does JamaicaVM support the Real-Time Specification for Java (RTSJ)?

A: Yes. The RTSJ V1.0.2 is supported by JamaicaVM 6.2. The API documentation of the implementation can be found at `http://www.aicas.com/jamaica/doc/rtsj_api/index.html`.

Q: The realtime behavior is not as good as I expected when using JamaicaVM. Is there a way to improve this?

A: If you are using a POSIX operating system, the best realtime behavior can be achieved when using the FIFO scheduling policy. Note that Linux requires root access to set a realtime scheduling policy. See § 10.5.3

Q: Is Linux a real-time operating system?

A: No. However, kernel patches exist which add the functionality for real-time behavior to a regular Linux system.

A.3.5 Remote Method Invocation (RMI)

Q: Does Jamaica support RMI?

A: RMI is supported by Jamaica 6.2.

JamaicaVM 6.2 uses dynamically generated stub and skeleton classes. So no previous call to `rmic` is needed to generate those.

If the Builder is used to create RMI server applications, the exported interfaces and implementation classes need to be included.

An example build file demonstrating the use of RMI with Jamaica is provided with the JamaicaVM distribution. See Tab. 2.4.

Q: How can I use RMI?

A: RMI applications often comprise two separate programs, a server and a client. A typical server program creates some remote objects, makes references to these objects accessible, and waits for clients to invoke methods on these objects. A typical client program obtains a remote reference to one or more remote objects on a server and then invokes methods on them. RMI provides the mechanism by which the server and the client communicate and pass information back and forth.

Like any other Java application, a distributed application built by using Java RMI is made up of interfaces and classes. The interfaces declare methods. The classes implement the methods declared in the interfaces and, perhaps, declare additional methods as well. In a distributed application, some implementations might reside in some Java virtual machines but not others. Objects with methods that can be invoked across Java virtual machines are called remote objects.

An object becomes remote by implementing a remote interface, which has the following characteristics:

- A remote interface extends the interface `java.rmi.Remote`.
- In addition to any application-specific exceptions, each method signature of the interface declares `java.rmi.RemoteException` in its throws clause,.

Using RMI to develop a distributed application involves these general steps:

1. Designing and implementing the components of your distributed application.

2. Compiling sources.
3. Making classes network accessible.
4. Starting the application.

First, determine your application architecture, including which components are local objects and which components are remotely accessible. This step includes:

- Defining the remote interfaces. A remote interface specifies the methods that can be invoked remotely by a client. Clients program to remote interfaces, not to the implementation classes of those interfaces. The design of such interfaces includes the determination of the types of objects that will be used as the parameters and return values for these methods. If any of these interfaces or classes do not yet exist, you need to define them as well.
- Implementing the remote objects. Remote objects must implement one or more remote interfaces. The remote object class may include implementations of other interfaces and methods that are available only locally. If any local classes are to be used for parameters or return values of any of these methods, they must be implemented as well.

  Implementing the clients. Clients that use remote objects can be implemented at any time after the remote interfaces are defined, including after the remote objects have been deployed.

Example source code demonstrating the use of Remote Method Invocation is provided with the JamaicaVM distribution. See § 2.4.

Q: Does JamaicaVM include tools like `rmic` and `rmiregistry` to develop RMI applications?

A: The `rmiregistry` tool is included in JamaicaVM and can be executed like this:

```
jamaicavm sun.rmi.registry.RegistryImpl
```

JamaicaVM 3.0 added support for the dynamic generation of stub classes at runtime, obviating the need to use the Java Remote Method Invocation (Java RMI) stub compiler `rmic` to pre-generate stub classes for remote objects.

A.3.6 OSGi

Q: Does JamaicaVM support OSGi?

A: Yes. JamaicaVM runs with the Prosyst OSGi Runtime, Apache Felix and Eclipse Equinox.

Q: How can I improve the performance of my OSGi application?

A: OSGi loads every bundle with a different class loader, so the bundles will be loaded and interpreted at runtime. If a bundle does not need to be updated at runtime, the class loading can be delegated to the class loader of the OSGi framework to use the compiled built-in classes (see § 5).

Add the affected bundle to the classpath when building the application. Set the `org.osgi.framework.bundle.parent` property to `framework` and pass the list of packages used by the bundle to the framework with the `org.osgi.framework.bootdelegation` property.

A.4 Target-Specific Issues

A.4.1 VxWorks

Q: When I load a built application I get errors of the dynamic linker `Undefined symbol:`.

A: This linker error occurs when the VxWorks system is configured without one or more of the following function sets listed in "Configuration of VxWorks ". Please check your VxWorks configuration (INCLUDE_* statements) in `target/config/all/configAll.h` in your VxWorks installation directory. Probably you have to recompile a new VxWorks kernel image.

Q: When building on a Windows host system, many warnings of the following kind occur:

```
jamaica_native_io.o(.text+0x12): undefined reference to
'vprintf'
```

A: This problem is caused by a conflicting version of Cygwin being present on the system. The Builder expects the Cygwin version provided with the WindRiver Tools. In order to avoid these warnings, ensure that only the `cygwin1.dll` provided by the Tool Chain is loaded or on the path.

Q: On VxWorks 6.6 RTP or higher I observe a segmentation violation while executing a Jamaica virtual machine or a built application:

```
0x4529c6c (iJamaicavm_bin): RTP 0x452b010 has been stopped
due to signal 11.
```

A: This failure may be caused by one of several possible defects. Please make sure you use Jamaica 6.0 Release 2 or later. In addition, make sure that WindRiver's patches for bugs WIND00137239, WIND00151164 as well as WIND00225310 are installed on your VxWorks system.

WindRiver has confirmed WIND00151164 and WIND00225310 for VxWorks 6.6 and the x86 platform only. WIND00137239 was observed for VxWorks 6.8 x86 platform and VxWorks 6.7 PPC platform, but was confirmed for other platforms as well.

According to WindRiver, the presence of these patches can be confirmed by checking the version number reported by the C compiler. WindRiver recommended the following:

> In this case you can use the command `ccpentium -v` in VxWorks development shell, in the following directory:
>
> ```
> install_dir\gnu\4.1.2-vxworks-6.6\x86-win32\bin
> ```
>
> This will print the information about the GNU compiler that you need. The result should be:
>
> ```
> gcc version 4.1.2 (Wind River VxWorks G++ SJLJ-EH 4.1-238)
> ```

If there are difficulties in obtaining the patches or resolving the issue, please contact the aicas support team.

Q: On VxWorks RTP, versions 6.6 to 6.8, I observe an assertion failure while executing a Jamaica virtual machine or a built application:

```
In function _rtld_digest_phdr { headers.c:312 nsegs == 2
{ assertion failed
```

A: This failure is caused by the WindRiver bug WIND00137239. Please install the WindRiver patch for bug WIND00137239 or the GNU 4.1.2 Cumulative Patch for your VxWorks version and platform.

Q: On VxWorks 6.7 RTP I observe an exception in the task tNet0 while executing a Jamaica virtual machine or a built application which uses java.net:

```
0x169f020 (tNet0): task 0x169f020 has had a failure
and has been stopped.
```

A: This failure is caused by the WindRiver bug WIND00157790. Please install the WindRiver patch for bug WIND00157790 or the Service Pack 1 for VxWorks 6.7.1 and VxWorks Edition 3.7 Platforms. Then rebuild the VxWorks image. If you use a built application rebuild the application as well.

Appendix B

Information for Specific Targets

This appendix contains target specific documentation and descriptions.

B.1 Operating Systems

B.1.1 VxWorks

VxWorks from Wind River Systems is a real-time operating system for embedded computers. The JamaicaVM is available for VxWorks 5.4 to 6.9 and the following target hardware:

- ARM
- PowerPC
- SuperH4
- x86

B.1.1.1 Configuration of VxWorks

For general information on the configuration of VxWorks, please refer to the user documentation provided by WindRiver. For Jamaica, VxWorks should be configured to include the following functionality:[1]

- `INCLUDE_ATA`
- `INCLUDE_DEBUG_SHELL_CMD`
- `INCLUDE_DISK_UTIL_SHELL_CMD`

[1] Package names refer to VxWorks 6.6, names for other versions vary.

- INCLUDE_EDR_SHELL_CMD
- INCLUDE_GNU_INTRINSICS
- INCLUDE_HISTORY_FILE_SHELL_CMD
- INCLUDE_IPTELNETS
- INCLUDE_IPWRAP_GETIFADDRS
- INCLUDE_KERNEL_HARDENING
- INCLUDE_LOADER
- INCLUDE_NETWORK
- INCLUDE_NFS_CLIENT_ALL
- INCLUDE_NFS_MOUNT_ALL
- INCLUDE_PING
- INCLUDE_POSIX_SEM
- INCLUDE_POSIX_SIGNALS
- INCLUDE_SHELL
- INCLUDE_SHELL_EMACS_MODE
- INCLUDE_SHOW_ROUTINES
- INCLUDE_STANDALONE_SYM_TBL
- INCLUDE_STARTUP_SCRIPT
- INCLUDE_ROUTECMD
- INCLUDE_RTL8169_VXB_END
- INCLUDE_TASK_SHELL_CMD
- INCLUDE_TASK_UTIL
- INCLUDE_TC3C905_VXB_END
- INCLUDE_TELNET_CLIENT
- INCLUDE_UNLOADER

The module `INCLUDE_GNU_INTRINSICS` is only required if Jamaica was built using the GNU compiler, which is the default.

If VxWorks real time processes (aka RTP) are used, the following components are also required (RTPs generated with Jamaica are dynamically linked by default):

- `INCLUDE_POSIX_PTHREAD_SCHEDULER`
- `INCLUDE_SHL`
- `INCLUDE_RTP`
- `INCLUDE_RTP_SHELL_CMD`

If WindML graphics is used, it must be included as well:

- `INCLUDE_WINDML`

The number of available open files should be increased by setting the following parameters:

Parameter	**Value**	
`NUM_FILES`	1024	(DKM only)
`RTP_FD_NUM_MAX`	1024	(RTP only)

You might also need to set file system specific parameters. For example, if dosFs is used, then you'll also have to set the `DOSFS_DEFAULT_MAX_FILES` parameter.

In addition, the following parameters should be set:

Parameter	**Value**	
`TASK_USER_EXC_STACK_SIZE`	16384	

! If some of this functionally is not included in the VxWorks kernel image, linker errors may occur when loading an application built with Jamaica and the application may not run correctly.

B.1.1.2 Installation

The VxWorks version of Jamaica is installed as described in the section Installation (§ 2.1). In addition, the following steps are necessary.

Configuration for Tornado (VxWorks 5.x)

- Set the environment variable `WIND_BASE` to the base directory of the Tornado installation.
- We recommend you set the environment variable `WIND_BASE` in your boot- or login-script to the directory where Tornado is installed (top-level directory).
- Add the Tornado tools directory to the `PATH` environment variable, so that tools like `ccppc.exe` resp. `ccpentium.exe` can be found.

! Do not use the DOS/Windows-Style path separator "\" (backslash) in `WIND_BASE`, because some programs interpret the backslash as an escape sequence for special characters. Use "/" (slash) in path names.

Configuration of platform-specific tools (see § 2.1.1.2) is only required in special situations. Normally, setting the environment variable `WIND_BASE` and extending `PATH` is sufficient.

Configuration for Workbench (VxWorks 6.x)

- Set the environment variable `WIND_HOME` to the base directory of the WindRiver installation (e.g. `/opt/WindRiver`)
- Set the environment variable `WIND_BASE` to the VxWorks directory in the WindRiver installation. The previously declared environment variable `WIND_HOME` may be used (e.g., `$WIND_HOME/vxworks-6.6`).
- Set the environment variable `WIND_USR` to the RTP header files directory of the WindRiver installation (e.g., `$WIND_BASE/target/usr`).

We recommend using `wrenv.sh`, located in the WindRiver base directory to set all necessary environment variables. The VxWorks subdirectory has to be specified as the following example shows for VxWorks 6.6:

```
> /opt/WindRiver/wrenv.sh -p vxworks-6.6
```

! Do not add `wrenv.sh` to your boot or login script. It starts a new shell which tries to process its login-script and thus you create a recursion.

Configuration of platform-specific tools (see § 2.1.1.2) is only required in special situations. Normally, executing `wrenv.sh` is sufficient.

B.1.1.3 Starting an application (DKM)

The procedure for starting an application on VxWorks depends on whether downloadable kernel modules (DKM) or real-time processes (RTP) are used.

For DKM, if the target system is configured for disk, FTP or NFS access, simply enter the following command on the target shell:

```
-> ld < filename
```

Here, *filename* is the complete filename of the created application.

The main entry point address for an application built with the Jamaica Builder has the symbolic names "`jvm`" and "`jvm_`*destination*", where *destination* is either the name set via the Builder option `destination` or the name of the main class. For example, in the VxWorks target shell the HelloWorld application may be started with these commands:

```
-> sp jvm
-> sp jvm_HelloWorld
```

When starting an application that takes arguments, those are given in a single string as a second argument:

```
-> sp jvm,"args"
```

The start code of the created application parses this string and passes it as a standard Java string array to the main method. When starting a VM, all options and arguments must be put in this string according to the VM command line syntax.

Note: even if the Builder generates a file with the specified name, it may be renamed later, because the name of the main entry point is read from the symbol table included in the object file.

Setting environment variables Environment variables may be set in the VxWorks shell via the `putenv` command:

```
-> putenv("VARIABLE=value")
```

In order to start a user task that inherits these variables from the shell, the task must be spawned with the `VX_PRIVATE_ENV` bit set. To do so, use the `taskSpawn` command:

```
-> taskSpawn "jamaica",0,0x01000080,0x020000,jvm,"args"
```

Running two Jamaica applications at the same time In order to run two Jamaica applications at the same time, matching of common symbols by the kernel must be switched off. This is achieved by setting the global VxWorks variable `ldCommonMatchAll` to false prior to loading the applications.

```
-> ldCommonMatchAll=0
-> ld < RTHelloWorld
-> ld < HelloWorld
-> sp jvm_RTHelloWorld
-> sp jvm_HelloWorld
```

In the example, if `ldCommonMatchAll` were not set to `0`, HelloWorld would reuse symbols defined by RTHelloWorld.

Note that this functionality is not available on all versions of VxWorks. Please check the VxWorks kernel API reference.

Restarting a Jamaica application To restart a Jamaica application after it has terminated, it should be unloaded with the `unld` command and then reloaded. This is illustrated in the following example:

```
-> ld < HelloWorld
value = 783931720 = 0x2eb9d948 = 'H'
-> sp jvm_HelloWorld
[...]
-> unld 783931720
value = 0 = 0x0
-> ld < HelloWorld
value = 784003288 = 0x2ebaf0d8 = 'K'
-> sp jvm_HelloWorld
[...]
```

Note that the application should not be unloaded while still running. The `unld` command is optional, and the VxWorks image needs to be configured to include it by adding `INCLUDE_UNLOADER` to the configuration as suggested in § B.1.1.1.

B.1.1.4 Starting an application (RTP)

If real-time processes (aka RTP) are used, the dynamic library `libc.so` must be renamed to `libc.so.1` and added to the folder of the executable. This library is located in the WorkBench installation

`$WIND_BASE/target/usr/lib/`*arch*`/`*variant*`/common[le]/libc.so`

or (for VxWorks 6.8 and later)

`$WIND_BASE/target/lib[_smp]/usr/lib/`*arch*`/`*variant*`/common[le]/libc.so`

where, in case of an x86 architecture, *arch* is, for example, `pentium` and *variant* is, for example, `PENTIUM`. The `lib_smp` directory contains multicore libraries.

To start the application, please use the following shell command:

```
-> rtpSp "filename"
```

If you would like to specify command line parameters, add them as a space-separated list in the following fashion:

```
-> rtpSp "filename arg1 arg2 arg3"
```

The `rtpSp` command will pass environment variables from the shell to the spawned process.

B.1.1.5 Linking the application to the VxWorks kernel image

The built application may also be linked directly to the VxWorks kernel image, for example for saving the kernel and the application in FLASH memory. In the VxWorks kernel a user application can be invoked enabling the VxWorks configuration define `INCLUDE_USER_APPL` and defining the macro `USER_APPL_INIT` when compiling the kernel (see VxWorks documentation and the file `usrConfig.c`). The prototype to invoke the application created with the Builder is:

```
int jvm_main(const char *commandLine);
```

where *main* is the name of the main class or the name specified via the Builder option `destination`. To link the application with the VxWorks kernel image the macro `USER_APPL_INIT` should be set to something like this:

```
extern int jvm_main (const char *); jvm_main (args)
```

where *args* is the command line (as a C string) which should be passed to the application.

B.1.1.6 Enabling AltiVec on PowerPC Devices

If the PowerPC CPU of your target hardware supports AltiVec you can enable it for VxWorks DKM or RTP by setting the environment variable `JAMAICA_VXWORKS_ALTIVEC` to `true`.

B.1.1.7 Limitations

The current release of Jamaica for VxWorks has the following limitations:

- `java.lang.Runtime.exec()` is not implemented
- The method `java.lang.System.getenv()` that takes no parameters and returns a `java.util.Map` is not implemented.
- Loading of dynamic libraries at runtime is not supported. These methods are not implemented:
  - `System.loadLibrary(String)`
  - `Runtime.loadLibrary(String)`
  - `Runtime.load(String)`
- The following realtime signals are not available:

  `SIGSTKFLT`, `SIGURG`, `SIGXCPU`, `SIGXFSZ`, `SIGVTALRM`, `SIGPROF`, `SIGWINCH`, `SIGIO`, `SIGPWR`, `SIGSYS`, `SIGIOT` , `SIGUNUSED`, `SIGPOLL`, `SIGCLD`.
- Jamaica does not allow an application to set the resolution of the clock in `javax.realtime.RealtimeClock`. The resolution of the clock depends on the frequency of the system ticker (see `sysClkRateGet()` and `sysClkRateSet()`). If a higher resolution for the realtime clock is needed the frequency of the system ticker must be increased. Care must be taken when doing this, because other programs running on the system may change their behavior and even fail. In addition, under VxWorks 5.4 the realtime clock must be informed about changes of the system ticker rate with the function `clock_setres()`. The easiest way to do this is to add the following into a startup script for VxWorks

  ```
  sysClkRateSet(1000)
  timeSpec=malloc(8)
  (*(timeSpec+0))=0
  (*(timeSpec+4))=1000000
  clock_setres(0,timeSpec)
  free(timeSpec)
  ```

  This example sets the system ticker frequency to 1000 ticks per second and the resolution of the realtime clock to 1ms.

- Depending on the file system, `File.canRead()`, `File.canWrite()` and `File.canExecute()` may return incorrect values. These functions work for NFS, they do not necessarily work for local disk (FAT) and FTP. The reason for this limitation rooted in the implementation of `access()` provided by VxWorks.

- Depending on the file system, `RandomAccessFile.setLength()` may not work. This function works for local disk (FAT), they do not work for NFS. This is caused by the implementation of ioctl FIOTRUNC.

- Support for memory mapped buffers (`javax.nio`) is not available on VxWorks 5.x. This is due to `mmap` being unavailable.

- For parallel applications on VxWorks SMP the option `-Xcpus` can either be set to all CPUs or one CPU. Any other set of CPUs is currently not supported by VxWorks.

B.1.1.8 Additional notes

- Object files: because applications for VxWorks (DKM only) are usually only partially linked, missing external functions and missing object files cannot be detected at build time. If native code is included in the application with the option `object`, Jamaica cannot check at build time if all needed native code is linked to the application. This is only possible in the final linker step when the application is loaded on the target system.

B.1.2 Windows

B.1.2.1 Limitations

The current release of Jamaica for the desktop versions of Windows contains the following limitations:

- No realtime signals are available.

- Paths handled by `java.io.File` cannot be longer than 248 characters. This figure refers to the absolute path — that is, it is for example not possible to extend an absolute path of 240 characters by a relative path of 20 characters.

- On multicore systems Jamaica will always run on the first CPU in the system.

- The method `java.lang.System.getenv()` that takes no parameters and returns a `java.util.Map` is only implemented for Windows versions that support Unicode. To be precise: it is required that the system header file `windows.h` defines the `UNICODE` flag.

B.1.3 WindowsCE

B.1.3.1 Limitations

The current release of Jamaica for WindowsCE contains the following limitations:

- WindowsCE Version 5 limits process memory to 32MB of RAM. Therefore the application executable plus the amount of native stack for all threads in the thread pool plus the amount of memory required to display graphics must be less than 32MB.
- It is not possible to redirect the standard IO for processes created with `Runtime.exec()`.
- WindowsCE does not support the notion of a current working directory. All relative paths are interpreted as relative to the device root. Any file operation should be done with absolute paths. The methods `File.getAbsolutePath()` and `File.getCanonicalPath()` will prepend the value of the system property `user.dir`. Note that none of the other methods in `File` will honor `user.dir`. When setting `user.dir` it is important not to set it to a single backslash to avoid creating UNC paths. Instead `user.dir` should be set to ("`\.`") (which is the default setting in Jamaica)
- WindowsCE does not support environment variables. If you have a registry editor on your target, you can create string entries with the name of the environment variable in

  ```
  HKEY_CURRENT_USER\Software\aicas\jamaica\environment
  ```

  to use the variable in Jamaica.
- The method `java.lang.System.getenv()` that takes no parameters and returns a `java.util.Map` is not implemented.
- File locking through `FileChannel.lock()` is not supported for all file systems on WindowsCE. If WindowsCE does not support file locking for a given file system calls to FileChannel.lock() will fail silently. In particular, the UNC network filesystems does not support this mechanism.

- On `SH4` processors, JamaicaVM Builder uses C compiler optimization level `-Od` (no optimization) for all three optimization levels (size, speed, all). This is due to a bug in the C compiler (VisualStudio 2008 (9) — Microsoft (R) C/C++ Optimizing Compiler Version 15.00.20720 for Renesas SH).

- If the `UTF8` code page is not supported by the WindowsCE image, classes cannot be loaded dynamically. The VM will terminate with the message that `Unicode strings cannot be created`. System calls like reading a file might also fail.

B.1.4 OS-9

B.1.4.1 Limitations

The current release of Jamaica for OS-9 contains the following known limitations:

- The method `java.lang.System.getenv()` that takes no parameters and returns a `java.util.Map` is not implemented.

- Loading of dynamic libraries at runtime is not supported. These methods are not implemented:

  - `System.loadLibrary(String)`
  - `Runtime.loadLibrary(String)`
  - `Runtime.load(String)`

- `java.net.Socket.connect()`
  does not support a timeout. The value is ignored.

- `java.net.Socket.bind()`
  does not throw an exception if called several times with the same address.

- `java.nio.FileChannel.map()`
  is not supported.

B.1.5 QNX

B.1.5.1 Installation

To use the QNX toolchain, ensure that the following environment variables are set correctly (should be done during QNX installation):

- `QNX_HOST` (e.g., `C:/Programs/QNX632/host/win32/x86`)

- `QNX_TARGET` (e.g., `C:/Programs/QNX632/target/qnx6`)

For QNX 6.4 (and higher) the linker must be in the system path. On Linux, you can set this with the PATH environment variable:

```
export PATH=$PATH:/opt/QNX640/host/linux/x86/usr/bin
```

B.1.5.2 Limitations

Due to incorrect treatment of denormal double floating point values by floating point units for the ARM architecture, JamaicaVM for QNX on ARM uses soft floats by default. See also § B.2.1.1.

B.2 Processor Architectures

B.2.1 ARM

B.2.1.1 Use of the Floating Point Unit

Floating point units currently available for ARM processors do not fully support arithmetic conforming to IEEE 754 for the double format (64 Bit). So called *denormal* values may be treated as zero, which can lead to faulty results and non-termination of numerical algorithms, including algorithms in the Jamaica runtime libraries. On ARM it is therefore strongly recommended to use a library (*soft floats*) conforming to IEEE 754.

Whether denormal values are treated incorrectly can easily be identified with the Java program from Fig. B.1. The correct output for this program is as follows.

```
> jamaicac Denormal.java
> jamaicavm Denormal
Smallest normal value is 2.2250738585072014E-308
Largest denormal value is 2.225073858507201E-308
Their sum is 4.4501477170144023E-308
Expected value for the sum is 4.4501477170144023E-308
```

If denormal values are not supported, the sum instead is, incorrectly, again the smallest normal value, `2.2250738585072014E-308`.

```
public strictfp class Denormal
{
  // Hexadecimal representation of floating point values
  // Smallest normal
  private static final long NORM =   0x0010000000000000L;
  // Largest denormal
  private static final long DENORM = 0x000FFFFFFFFFFFFFL;
  // Their sum
  private static final long SUM =    0x001FFFFFFFFFFFFFL;

  public static void main(String[] args)
  {
    System.out.println("Smallest normal value is " +
                       Double.longBitsToDouble(NORM));
    System.out.println("Largest denormal value is " +
                       Double.longBitsToDouble(DENORM));
    System.out.println("Their sum is " +
                       (Double.longBitsToDouble(NORM) +
                        Double.longBitsToDouble(DENORM)));
    System.out.println("Expected value for the sum is " +
                       Double.longBitsToDouble(SUM));
  }
}
```

Figure B.1: `Denormal.java` — Identify whether denormal floating point values are treated correctly.

Appendix C

Heap Usage for Java Datatypes

This chapter contains a list of in-memory sizes of datatypes used by JamaicaVM.

For datatypes that are smaller than one machine word, only the smallest multiple of eight Bits that fits the datatype will be occupied for the value. I.e., several values of types boolean, byte, short and char may be packed into a single machine word when stored in an instance field or an array.

Tab. C.1 shows the usage of heap memory for primitive types, Tab. C.2 shows the usage of heap memory for objects, arrays and frames.

Datatype	Used Memory		Min Value	Max Value
	Bits	Bytes		
`boolean`	8	1	-	-
`byte`	8	1	-2^7	$2^7 - 1$
`short`	16	2	-2^{15}	$2^{15} - 1$
`char`	16	2	\u0000	\uffff
`int`	32	4	-2^{31}	$2^{31} - 1$
`long`	64	8	-2^{63}	$2^{63} - 1$
`float`	32	4	1.4E-45F	3.4028235E38F
`double`	64	8	4.9E-324	1.7976931348623157E308
Java reference				
32-bit systems	32	4	-	-
64-bit systems	32	4	-	-

Table C.1: Memory Demand of Primitive Types

Data Structure	Memory Demand
Object header (containing garbage collection state, object type, inlined monitor and memory area)	12 Bytes
Array header (containing object header, array layout information and array length)	16 Bytes
Java object size on heap (minimum)	32 Bytes
Java array size on heap (minimum)	32 Bytes
Minimum size of single heap memory chunk	64 KBytes
Garbage Collector data overhead for heap memory. For a usable heap of a given size, the garbage collector will allocate this proportion of additional memory for its data.	
Single-core systems	6.25%
Multi-core, 32-bit systems	15.63%
Multi-core, 64-bit systems	18.75%
Stack slot	8 Bytes
Java stack frame of normal method	4 slots
Java stack frame of synchronized method	5 slots
Java stack frame of static initializer	7 slots
Java stack frame of asynchronously interruptible method	8 slots
Additional Java stack frame data in profile mode	2 slots

Table C.2: Memory Demand of Objects, Arrays and Frames

Appendix D

Limitations

This appendix lists limitations of the JamaicaVM virtual machine and applications created with JamaicaVM builder.

D.1 VM Limitations

These limitations apply to both pre-built virtual machines and to applications built with the JamaicaVM builder.

- Numeric limitations, such as the absolute maximum number of Java Threads or the absolute maximum heap size are listed in Tab. D.1.

Aspect	**Limit**
Number of Java Threads	511
Maximum Monitor Nest Count (repeated monitor enter of the same monitor in nested `synchronized` statements or nested calls to `synchronized` methods). Exceeding this value will result in throwing an `java.lang.InternalError` with detail message `"Max. monitor nest count reached (255)"`	255

Aspect	Limit
Minimum Java heap size	64KB
Maximum Java heap size (32-bit systems)	approx. 3.5GB
Maximum Java heap size (64-bit systems)	approx. 120GB
Minimum Java heap size increment	64KB
Maximum number of heap increments. The Java heap may not consist of more than this number of chunks, i.e., when dynamic heap expansion is used (max heap size is larger than initial heap size), no more than this number of increments will be performed, including the initial chunk. To avoid this limit, the heap size increment will automatically be set to a larger value when more than this number of increments would be needed to reach the maximum heap size.	256
Maximum number of memory areas (instances of `javax.realtime.MemoryArea`). Note that two instances are used for `HeapMemory` and `ImmortalMemory`.	256
Maximum size of Java stack	64MB
Maximum size of native stack	2GB
Maximum number of constant UTF8 strings (names and signatures of methods, fields, classes, interfaces and contents of constant Java strings) in the global constant pool (exceeding this value will result in a larger application)	$2^{24} - 1$
Maximum number of constant Java strings in the global constant pool (exceeding this value will result in a larger application)	$2^{16} - 1$
Maximum number of name and type entries (references to different methods or fields) in the global constant pool (exceeding this value will result in a larger application)	$2^{16} - 1$
Maximum Java array length. Independent of the heap size, Java arrays may not have more than this number of elements. However, the array length is not restricted by the heap size increment, i.e., even a heap consisting of several increments each of which is smaller than the memory required for a Java array permits the allocation of arrays up to this length provided that the total available memory is sufficient.	$2^{27} - 1$

Aspect	Limit
Maximum number of virtual methods per Java class (including inherited virtual methods)	4095
Maximum number of interface methods per Java interface (including interface methods inherited from superinterface)	4095
On POSIX systems where `time_spec.tv_sec` is a 32 Bit value it is not possible to wait until a time and date that is later than	Tue Jan 19 04:14:07 2038

Table D.1: JamaicaVM limitations

D.2 Builder Limitations

The static compiler does not compile certain Java methods but leaves them in interpreted bytecode format independent of the compiler options or their significance in a profile.

- Static initializer methods (methods with name `<clinit>`) are not compiled.

  A simple way to enable compilation is to change a static initializer into a static method, which will be compiled. That is, replace a static initializer

```
class A
{
  static
    {
      <initialization code>
    }
}
```

  by the following code:

```
class A
{
  static
    {
      init();
    }
  private static void init()
    {
```

```
        <initialization code>
    }
}
```

- Methods with bytecode that is longer than the value provided by builder option `XexcludeLongerThan` are not compiled.
- When option `lazy` is set (which is the default), methods that reference a class, field or method that is not present at build time are not compiled. The referenced class will be loaded lazily by the interpreter.

D.3 Multicore Limitations

Currently, the multicore variant of the JamaicaVM virtual machines (command `jamaicavmm`) and the JamaicaVM builder using option `-parallel` have the following additional limitations.

- The class `javax.realtime.MonitorControl` is not supported.
- In class `com.aicas.jamaica.lang.Debug`, methods `getMaxFree-RangeSize`, `getNumberOfFreeRanges`, `printFreeListStats` and `createFreeRangeStats` are not supported.
- Java arrays that are not allocated very early during application startup (before the garbage collector starts recycling memory) are allocated using a non-contiguous representation that results in higher costs for array accesses.
- The multicore VM does not support the JVMTI interface. In particular, the option `-agentlib` of both the VM and the builder does not work.

Appendix E

Internal Environment Variables

Additional debugging output can be activated through environment variables if an application was built with the internal option `-debug=true`. This option and its environment variables are used for debugging Jamaica itself and are not normally relevant for users of JamaicaVM.

JAMAICA_DEBUGLEVEL Defines the debug level of an application that was built with the option `debug`. A level of 0 means that only a small amount of debug output is printed; a level of 10 means that very detailed debug output is printed.

Note that at a debug level of 10 a simple HelloWorld application will produce thousands of lines of debug output. A good choice is a level of about 5.

JAMAICA_DEBUGCALLNATIVE Defines a string that gives the name of a native method. Any call to that method is printed in addition to other debug output. Printing of these calls requires a minimum debug level of 5. If the variable is not set or set to '*', any native call will be printed.

JAMAICA_DEBUGCALLJAVA Defines a string that gives the name of a Java class or method. Any call to the specified method or to a method defined in the specified class will be printed in addition to the other debug output.

Printing of these calls requires a minimum debug level of 5. If the variable is not set or set to '*', any call is printed. E.g., setting `JAMAICA_DEBUGCALLJAVA` to `java/lang/String.length` will print any call to the method `java.lang.String.length()`.

Bibliography

[1] Stephane Bailliez, Nicola Ken Barozzi, Jacques Bergeron, Stefan Bodewig, Patrick Chanezon, James Duncan Davidson, Tom Dimock, Peter Donald, Dion Gillard, Erik Hatcher, Diane Holt, Bill Kelly, Martijn Kruithof, Arnout J. Kuiper, Antoine Lévy-Lambert, Conor MacNeill, Jan Matèrne, Stefano Mazzocchi, Erik Meade, Sam Ruby, Nico Seessle, Jon S. Stevens, Wolf Siberski, Magesh Umasankar, Roger Vaughn, Dave Walend, Phillip Wells, Christoph Wilhelms, and Craeg Strong. Apache Ant™ manual. URL: `http://ant.apache.org/manual/`.

[2] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, 2000.

[3] Peter C. Dibble. *Real-Time Java Platform Programming*. Prentice-Hall, 2002.

[4] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, Boston, Mass., third edition, 2005.

[5] Mike Jones. What really happened on Mars? URL: `http://research.microsoft.com/~mbj/Mars_Pathfinder/`, 1997.

[6] Sheng Liang. *Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley, 1999.

[7] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, second edition, 1999.

[8] C. L. Liu and J. W. Wayland. Scheduling algorithms for multiprogramming in hard real-time environment. *Journal of the ACM*, 20, 1973.

[9] Fridtjof Siebert. Concurrent, parallel, real-time garbage-collection. In *ACM Sigplan Notices*, volume 45, pages 11–20, 2010.

Index of Options

-?
  builder, 160
  jamaicah, 206
  vm, 142

-agentlib
  builder, 160
  vm, 147
-analyse
  builder, 178
-analyseFromEnv
  builder, 179
-analyze
  builder, 178
-analyzeFromEnv
  builder, 179
-atomicGC
  builder, 180

-bootclasspath
  jamaicac, 138
  jamaicah, 207

-classname
  jamaicah, 207
-classpath
  builder, 161
  jamaicah, 207
  vm, 142
-closed
  builder, 167
-compile
  builder, 169
-configuration
  builder, 161
-constGCwork
  builder, 179
-constGCworkFromEnv
  builder, 180
-cp
  builder, 161
  jamaicac, 137
  jamaicah, 207

-D
  vm, 142
-d
  jamaicac, 138
  jamaicah, 207
-deprecation
  jamaicac, 138
-destination
  builder, 165
-dwarf2
  builder, 188

-ea
  builder, 161
-enableassertions
  builder, 161
-encoding
  jamaicac, 139
-excludeClasses
  builder, 162
-excludeFromCompile
  builder, 163
-excludeJAR
  builder, 164

-extdirs
jamaicac, 138

-finalizerPri
builder, 175
-finalizerPriFromEnv
builder, 176

-g
jamaicac, 139

-h
builder, 160
jamaicah, 206
-heapSize
builder, 171
-heapSizeFromEnv
builder, 173
-heapSizeIncrement
builder, 172
-heapSizeIncrementFromEnv
builder, 173
-help
builder, 160
jamaicah, 206
vm, 142
--help
builder, 160
jamaicah, 206

-immortalMemorySize
builder, 183
-immortalMemorySizeFromEnv
builder, 183
-includeClasses
builder, 162
-includeFilename
jamaicah, 207
-includeInCompile
builder, 163
-includeJAR
builder, 164
-incrementalCompilation
builder, 167
-inline
builder, 170
-interpret
builder, 169

-J
jamaicac, 139
-jar
builder, 162
-javaStackSize
builder, 172
-javaStackSizeFromEnv
builder, 173
-jni
jamaicah, 207
-js
vm, 144

-lazy
builder, 164
-lazyFromEnv
builder, 165

-main
builder, 161
-maxHeapSize
builder, 171
-maxHeapSizeFromEnv
builder, 173
-maxNumThreads
builder, 174
-maxNumThreadsFromEnv
builder, 176
-mi
vm, 144
-ms
vm, 144
-mx
vm, 144

-nativeStackSize
builder, 172
-nativeStackSizeFromEnv
builder, 173
-nowarn
jamaicac, 138
-ns
vm, 144
-numJniAttachableThreads
builder, 174
-numJniAttachableThreadsFromEnv
builder, 176
-numThreads
builder, 174
-numThreadsFromEnv
builder, 176

-o
builder, 165
jamaicah, 207
-object
builder, 184
-optimise
builder, 171
-optimize
builder, 171

-parallel
builder, 178
-percentageCompiled
builder, 170
-physicalMemoryRanges
builder, 183
-priMap
builder, 176
-priMapFromEnv
builder, 177
-profile
builder, 170

-reservedMemory
builder, 180
-reservedMemoryFromEnv
builder, 181
-resource
builder, 165

-saveSettings
builder, 161
-schedulingPolicy
builder, 178
-schedulingPolicyFromEnv
builder, 178
-scopedMemorySize
builder, 183
-scopedMemorySizeFromEnv
builder, 183
-setFonts
builder, 166
-setGraphics
builder, 166
-setLocalCryptoPolicy
builder, 166
-setLocales
builder, 166
-setProtocols
builder, 167
-setTimeZones
builder, 166
-showExcludedFeatures
builder, 168
-showIncludedFeatures
builder, 168
-showNumberOfBlocks
builder, 169
-showSettings
builder, 161
-smart
builder, 167
-source
jamaicac, 138
-sourcepath
jamaicac, 138

-ss
vm, 144
-stopTheWorldGC
builder, 180
-strictRTSJ
builder, 181
-strictRTSJFromEnv
builder, 182

-target
builder, 171
jamaicac, 138
-threadPreemption
builder, 175
-timeSlice
builder, 175
-timeSliceFromEnv
builder, 175
-tmpdir
builder, 165

-useProfile
builder, 170
-useTarget
jamaicac, 137

-verbose
builder, 160
-version
builder, 160
jamaicah, 207
vm, 142

-X
jamaicac, 139
vm, 142
-XavailableTargets
builder, 189
-Xbootclasspath
builder, 185
jamaicah, 207
vm, 143
-Xbootclasspath/a
vm, 143
-Xbootclasspath/p
vm, 143
-Xcc
builder, 187
-XCFLAGS
builder, 187
-Xcheck:jni
vm, 145
-Xcpus
builder, 189
vm, 144
-XcpusFromEnv
builder, 190
-XdefineProperty
builder, 184
-XdefinePropertyFromEnv
builder, 185
-XenableDynamicJNILibraries
builder, 190
-XexcludeLongerThan
builder, 187
-XfullStackTrace
builder, 187
-Xhelp
builder, 160
jamaicah, 206
--Xhelp
builder, 160
-xhelp
vm, 142
-XignoreLineNumbers
builder, 185
-Xinclude
builder, 191
-Xint
builder, 169
-XjamaicaHome
builder, 185
-XjavaHome

builder, 185
-Xjs
vm, 144
-XlazyConstantStrings
builder, 185
-XlazyConstantStringsFromEnv
builder, 186
-Xld
builder, 187
-XLDFLAGS
builder, 187
-Xlibraries
builder, 188
-XlibraryPaths
builder, 188
-XloadJNIDynamic
builder, 191
-Xmi
vm, 144
-Xms
vm, 144
-Xmx
vm, 144
-XnoClasses
builder, 186
-XnoMain
builder, 186
-XnoRuntimeChecks
builder, 189
-Xns
vm, 144
-XobjectFormat
builder, 191
-XobjectProcessorFamily
builder, 191
-XobjectSymbolPrefix
builder, 191
-Xprof
vm, 144
-XprofileFilename
builder, 186
vm, 147
-XprofileFilenameFromEnv
builder, 187
-Xss
vm, 144
-XstaticLibraries
builder, 188
-Xstrip
builder, 188
-XstripOptions
builder, 188
-XuseMonotonicClock
builder, 190
-XuseMonotonicClockFromEnv
builder, 190